Hochschule Offenburg
University of Applied Sciences

Leistungsanalyse und Optimierung eines Flugdatenservers

Bachelor-Thesis
zur Erlangung des akademischen Grades „Bachelor of Science“
WS 2011/2012

Fakultät Elektro- und Informationstechnik
Studiengang Angewandte Informatik

Vorgelegt von: Francois Hilberer
Karolingerstraße 7
76137 Karlsruhe

Betreuer: Prof. Dr. rer. nat. Erwin Mayer
Prof. Dr. rer. nat. Klaus Dorer

Eingereicht am 31.10.2011

Kurzfassung

Diese Arbeit beschäftigt sich mit der Leistungsanalyse und Optimierung eines Flugdatenservers. Für die Durchführung dieser Leistungsanalyse wird eine eigene Anwendung entwickelt. Mit dieser eigens entwickelten Anwendung wird das Leistungsverhalten des Flugdatenservers genau analysiert. Mit den Erkenntnissen aus der Leistungsanalyse werden am Flugdatenserver Optimierungen durchgeführt und weitere Möglichkeiten zur Optimierung aufgezeigt.

Bei diesem Flugdatenserver (Air Traffic Control Server) handelt es sich um eine Client/Server-Anwendung zur Visualisierung von Flugbewegungen im Lauftraum. Hierfür wird der Flugdatenserver mit Flugsicherungsdaten durch das Flugüberwachungssystem Automatic Dependent Surveillance (ADS) versorgt.

Vorwort

Ich möchte mich bei allen Personen bedanken die mich während meines Studiums unterstützt haben. Insbesondere bei meinen Eltern und meiner Lebensgefährtin. Des Weiteren möchte ich mich bei meinem Erstbetreuer Prof. Dr. Erwin Mayer und bei meinem Zweitbetreuer Prof. Dr. Klaus Dorer für die ausgezeichnete Betreuung und Unterstützung bedanken.

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelor-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Offenburg, den 31.10. 2011

_____

Unterschrift (Francois Hilberer)

Abkürzungsverzeichnis

ATC	Air Traffic Control
ADS	Automatic Dependent Surveillance
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASTERIX	All Purpose Structured Eurocontrol Surveillance Information Exchange
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IPC	Inter Process Communication
ISC	Inter Server Communication
JSON	JavaScript Object Notation
REST	Representational State Transfer
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
XML	Extensible Markup Language

Inhaltsverzeichnis

1 Einführung

1.1 Ausgangssituation

Grundlage für diese Bachelor-Thesis ist der im Rahmen der Lehrveranstaltung *Projekt 2* im 6. Semester im Studiengang Angewandte Informatik entwickelte Flugdatenserver (*Air Traffic Control Server)*. Dieser Flugdatenserver erhält Flugsicherungsdaten von sogenannten ADS-B Antennen. Bei diesen ADS-B Antennen handelt es sich um Empfangsstationen bzw. Bodenstationen des Flugüberwachungssystems *Automatic Dependent Surveillance (ADS).*

Bei diesem Flugüberwachungssystem sind Verkehrsflugzeuge mit einem Transponder ausgestattet, der kontinuierlich ohne Aufforderung (unabhängig) Flugzeugnachrichten (ADS-B Nachrichten) an alle ADS-B Empfangsstationen (Broadcast) die sich in Reichweite befinden sendet. Die ADS-B Antennen Bodenstationen wiederum leiten die erhaltenen Flugzeugnachrichten an registrierte Hosts weiter. ADS-B dient allerdings nicht nur zur Übertragung an Bodenstationen sondern auch zur Übertragung zwischen den Flugzeugen selbst.

Der Flugdatenserver der Hochschule Offenburg ist momentan an drei solcher ADS-B Antennen registriert und erhält somit von diesen Flugzeugnachrichten. Einer dieser Antennen befindet sich auf dem Gebäude der Hochschule Offenburg. Die empfangenen Flugzeugnachrichten enthalten unter anderem Informationen über die Identität des Flugzeuges, aktuelle Position, Höhe und Geschwindigkeit.

Grundsätzlich hat der Flugdatenserver die Aufgabe diese Flugzeugnachrichten zu empfangen, diese zu verarbeiten und in einer Web-Anwendung in Form von Flugbewegungen im Luftraum zu visualisieren.

1.2 Motivation

Die Web-Anwendung des Flugdatenservers zur Visualisierung von Flugbewegungen, soll im Rahmen der Internetpräsenz (Webseite) der Hochschule Offenburg einem möglichst großen Zielpublikum zur Verfügung gestellt werden. Mit dieser Bachelor-Thesis soll festgestellt werden,

inwiefern der Flugdatenserver eine größere Anzahl an Benutzer dieser Web-Anwendung verarbeiten kann bzw. unterstützt.

Eine Web-Anwendung ist potenziell für Millionen von Benutzern über das Internet erreichbar. Deshalb ist es wichtig festzustellen, inwieweit die eigene Web-Anwendung eine größere Menge an Benutzer verarbeiten kann. Typische Probleme einer Web-Anwendung sind die Gewährleistung ihrer eigenen Verfügbarkeit und ihr Zeitverhalten hinsichtlich der Verarbeitung von Client-Anfragen mit einer steigenden Benutzerlast. So können ab einer bestimmten Benutzerlast Leistungsengpässe entstehen z.B. in Form von langen Antwortzeiten auf die Client-Anfragen. Diese Schwelle der Benutzerlast ist natürlich abhängig von der jeweiligen Web-Anwendung.

Bei der Visualisierung von Flugbewegungen ist es aber wichtig, dass kontinuierlich neue Flugzeugnachrichten empfangen und die Visualisierung zyklisch aktualisiert wird. Um dies zu gewährleisten sind schnelle und häufige Client-Anfragen zur Aktualisierung notwendig.

1.3 Zielbestimmung

Es soll evaluiert werden, inwiefern die Web-Anwendung des Flugdatenservers einer breiten Masse an Benutzer zur Verfügung gestellt werden kann. Mit einer Leistungsanalyse des Flugdatenservers sollen mögliche Leistungsengpässe ermittelt werden. Auf Basis der Ergebnisse dieser Leistungsanalyse sollen Optimierung vorgenommen werden, um die Leistungsengpässe zu minimieren bzw. zu beheben.

Die Leistungsanalyse selbst soll mit Hilfe von simulierten Zugriffen auf den Flugdatenserver realisiert werden. Mit dieser Simulation der Zugriffe soll das Verhalten des Flugdatenservers bei zunehmender Nutzerzahl genau analysiert werden. Für die Simulation von Zugriffen auf Web-Anwendungen, gibt es bereits ein vielfältiges Angebot an so genannten „*Load Testings Tools*“. Aufgrund von verschiedenen domänenspezifischen Eigenschaften und der Möglichkeit flexibel eigene Anforderungen zu integrieren, soll für die Simulation der Zugriffe eine eigene Anwendung entwickelt und für die Leistungsanalyse eingesetzt werden. Während einer solchen Simulation von Client-Zugriffen sollen zudem verschiedene Leistungsmetriken wie Antwortzeiten auf die simulierten Client-Anfragen sowie CPU-, Arbeitsspeicher- und Netzwerkschnittstellen-Auslastung des Flugdatenservers ermittelt werden. Anhand dieser Leistungsmetriken kann die Leistung des Flugdatenservers bewertet werden.

1.4 Aufbau dieser Arbeit

Die Arbeit gliedert sich in vier Teile:

- Grundlagen
- Entwicklung der Anwendung zur Leistungsanalyse
- Leistungsanalyse
- Optimierung

Im ersten Teil der Arbeit werden Grundlagen zur Flugüberwachung, zum Flugdatenserver und zum Thema „Leistungstests“ und „Optimierung“ erarbeitet. Der Begriff Flugüberwachung wird definiert und Möglichkeiten zur Flugüberwachung beschrieben. Der Flugdatenserver wird ausführlich beschrieben, insbesondere die Architektur und Kommunikation des Flugdatenservers mit den anfragenden Web-Clients. Anschließend wird auf das Thema Leistungstests und Optimierung eingegangen.

Im zweiten Teil dieser Arbeit wird die Entwicklung der Anwendung zur Leistungsanalyse beschrieben. Dabei werden die wichtigsten Anforderungen zur Anwendung erläutert, es werden verschiedene Entwurfsalternativen dargestellt, sowie eine ausführliche Beschreibung zum Entwurf und der Implementierung der Anwendung und Ihrer Komponenten.

Auf Basis des ersten und zweiten Teils der Thesis, wird im dritten Teil die Leistungsanalyse des Flugdatenservers beschrieben. Dabei wird insbesondere auf die Vorgehensweise und Ergebnisse zur Leistungsanalyse eingegangen.

Im letzten Teil dieser Arbeit werden anhand der Erkenntnisse aus der Leistungsanalyse Möglichkeiten aufgezeigt den Flugdatenserver zu optimieren.

2 Grundlagen

2.1 Flugüberwachung

In diesem Kapitel werden Grundlagen zur Flugüberwachung und zum Flugüberwachungssystem *Automatic Dependent Surveillance* (ADS) erläutert.

2.1.1 Einführung

Die Flugüberwachung ist verantwortlich für einen sichern, geordneten und flüssigen Luftraumverkehr. Um dies zu gewährleisten, ist die Überwachung des Lauftraumverkehrs von zentraler Bedeutung. Hierfür ist es wichtig Flugobjekte und deren Position im Lauftraum zu bestimmen, um einen Gesamtüberblick der Flugbewegungen im Luftraum zu erhalten. Für die Bestimmung von Flugobjekten und deren Position im Luftraum werden typischerweise Radarsysteme eingesetzt. Hier unterscheidet man zwischen Primär- und Sekundärradarsystemen. [WikFlug]

Beim Primärradarsystem senden Radaranlagen Elektromagnetische Wellen aus, die von möglichen Hindernissen im Luftraum zurück reflektiert werden. Mit dieser zurück reflektieren Energie (Echo) kann die Position des Flugobjektes im Luftraum bestimmt werden. Durch eine Laufzeitmessung der ausgestrahlten und reflektieren Signale und der Position der Antennen kann zudem die Entfernung und die Richtung des Flugobjektes ermittelt werden. Bei diesem Verfahren spricht man von einer passiven Funkortung, da das Flugobjekt selbst nicht aktiv dabei mitwirkt, sondern nur die Signale reflektiert. Mit dem Primärradar kann zwar die Position des Flugobjektes bestimmt werden, aber weitere relevante Informationen wie Höhe und Identität des Flugzeuges können nicht ermittelt werden. [MEN04] [WikPri]

Beim Sekundärradar senden Flugobjekte bei Erhalt eines Radarsignals selbst aktiv eine Antwort. In dieser Antwort wird die Identität und die Höhe des Flugobjektes an die Bodenstation übermittelt. Für das Empfangen und Senden eines Signals ist das Flugobjekt mit einem entsprechenden Transponder ausgestattet. [MEN04] [WigSek]

2.1.2 Automatic Dependent Surveillance (ADS)

Bei *Automatic Dependent Surveillance* (ADS) handelt es sich um ein weiteres Verfahren, welches zur Flugüberwachung eingesetzt wird. Bei diesem Verfahren senden Flugzeuge aktiv und ohne vorherige Aufforderdung Flugzeuginformationen an sämtliche Empfangsstationen die sich in Reichweite befinden. Das Senden erfolgt durch einen ADS-B Transponder der kontinuierlich, typischerweise im Intervall von einer Sekunde, Flugzeugnachrichten versendet. In diesen Flugzeugnachrichten sind alle relevanten Informationen wie Identität (Call Sign), Position, Geschwindigkeit, Höhe, Richtung etc. des Flugzeuges enthalten. Die Position (Längengrad und Breitengrad) bestimmt das Flugzeug dabei mittels GPS.

Da die Nachrichten broadcast und somit an keine speziellen Empfänger adressiert werden, wird das System auch als ADS-B (broadcast) bezeichnet. Empfangsstationen die diese Flugzeug-nachrichten erhalten, leiten diese an registrierte Hosts wie z.B. eine Flugüberwachungsstation weiter.

Grundsätzlich wird ADS-B zur Ortung und Überwachung von Luftfahrzeugen verwendet. Im Vergleich zum konventionellen Radarverfahren ist ADS-B kostengünstiger, da auf teure Radarstationen verzichtet werden kann. Gerade in Regionen wo keine flächendeckende Radarinfrastruktur vorhanden ist, ist die Nutzung von ADS-B sehr interessant. Geplant ist das bis 2012 nahezu alle Verkehrsflugzeuge mit einem ADS-B Transponder ausgestattet sind. In der nachfolgenden Abbildung 2.1.2.1 wird der ADS-B Informationsfluss dargestellt. [MEI04]

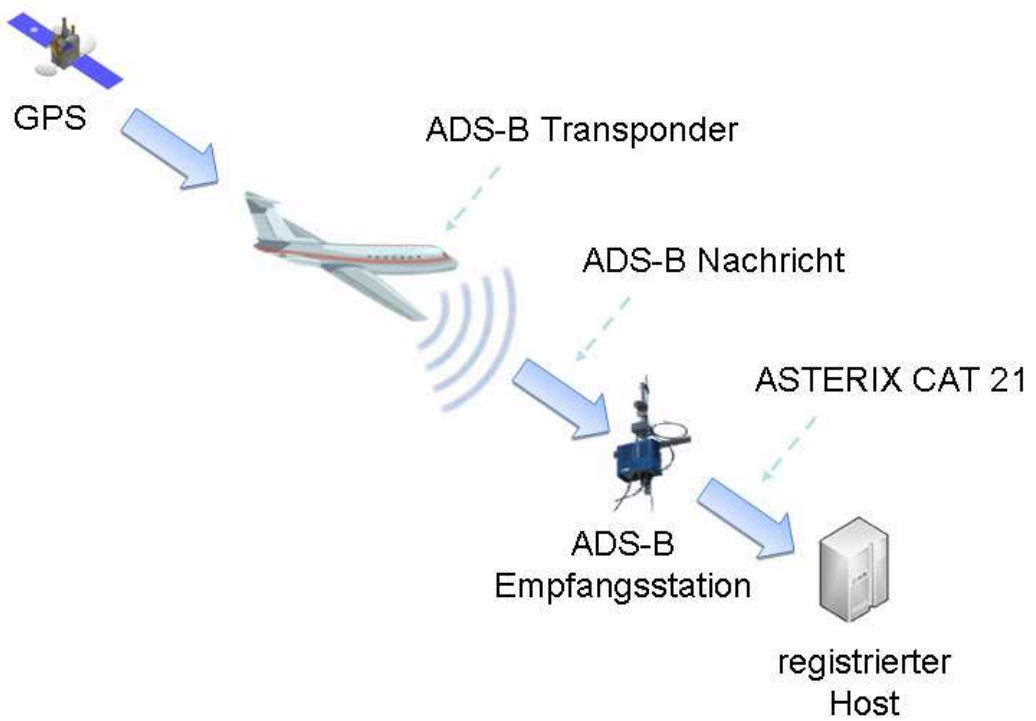


Abb. 2.1.2.1: Informationsfluss bei ADS-B.

2.1.3 ASTERIX Category 21

Bei ASTERIX Category 21 (All Purpose Structured Eurocontrol Surveillance Information Exchange) handelt es sich um eine Beschreibung, die den Aufbau und den Inhalt der ADS-B Nachrichten, die von der ADS-B Antenne weitergeleitet werden, festlegt. Diese Beschreibung bzw. dieser Kommunikationsstandard zur Datenübertragung wird von der Organisation EUROCONTROL vorgegeben. EUROCONTROL ist eine internationale Organisation zur Koordination der Luftverkehrskontrolle in Europa.

Die ASTERIX Cat 21 Daten werden grundsätzlich in einem Binärformat übertragen. Eine ASTERIX Cat 21 Nachricht kann beliebig viele Flugzeugnachrichten enthalten. Die Anzahl der Flugzeugnachrichten ist dabei abhängig von der jeweiligen ADS-B Antenne und deren Empfangsabdeckung. Das ASTERIX Cat 21 Format enthält zudem eine Vielzahl von optionalen Feldern, so dass eine exakte Größenangabe nicht möglich ist. [PAH09] [CAT21]

Nachfolgend werden die wichtigsten Informationen die aus dem ASTERIX Cat 21 Datenformat extrahiert werden können aufgelistet:

- Zeitstempel: Versende Zeitpunkt der Nachricht
- Call Sign: Identität des Flugzeuges
- Position des Flugobjektes nach Längen- und Breitengrad
- Höhe des Flugobjektes relativ zum Meeresspiegel in Metern
- Flugrichtung des Flugobjektes (vertikal und horizontal)
- Geschwindigkeit des Flugobjektes in Kilometern/Stunde

2.2 Flugdatenserver (Air Traffic Control Server)

In diesem Kapitel werden Grundlagen zum Flugdatenserver, dessen Architektur, die Kommunikationsschnittstellen und der Ablauf der Kommunikation beschrieben.

2.2.1 Einführung

Der Flugdatenserver wurde im Rahmen der Lehrveranstaltung *Projekt 2* im 6. Semester im Studiengang Angewandte Informatik in Teamarbeit entwickelt. Bei dieser Lehrveranstaltung werden Industrie-nahe Projekte simuliert und innerhalb eines Semesters umgesetzt. Das Team für dieses Projekt umfasste dabei neun Entwickler (Studenten). Als Basis für die Entwicklung des Flugdatenservers konnten Bestandteile aus einer bereits vorhandenen Desktop-Anwendung zur Visualisierung von Flugbewegungen verwendet werden. Diese Desktop-Anwendung wurde im Rahmen der Bachelor-Thesis von Herrn Simon Pahl im Wintersemester 2008/2009 entwickelt.

Bei dem Flugdatenserver *(Air Traffic Control Server)* handelt es sich um eine Client/Server-Anwendung zur Visualisierung von Flugbewegungen im Lauftraum. In der Server-Anwendung werden Flugzeugnachrichten empfangen, aufbereitet und gespeichert. Die Web-Client-Anwendung visualisiert die Flugzeugnachrichten, in Form von Flugzeugbewegungen, in einem Browser (Web-Anwendung).

Wie bereits beschrieben erhält der Flugdatenserver Flugsicherungsdaten von den ADS-B Antennen des Flugüberwachungssystems *Automatic Dependence Surveillance (ADS).* Derzeit erhält der Flugdatenserver kontinuierlich von drei solcher ADS-B Antennen ASTERIX Cat 21 Nachrichten. Eine dieser ADS-B Antennen befindet sich auf dem Gebäudekomplex der Hochschule Offenburg. Die zwei weiteren Antennen befinden sich in Karlsruhe und in Stuttgart. Die ADS-B Empfangsabdeckung des Flugdatenservers vereint somit drei große Empfangsbereiche und erreicht eine Reichweite, die sich von Frankfurt am Main bis weit über die Schweizer Grenze erstreckt.

In der nachfolgenden Abbildung 2.2.1.1 wird der Informationsfluss ausgehend vom Flugzeug (GPS-Satelliten) bis hin zu den Web-Clients dargestellt.

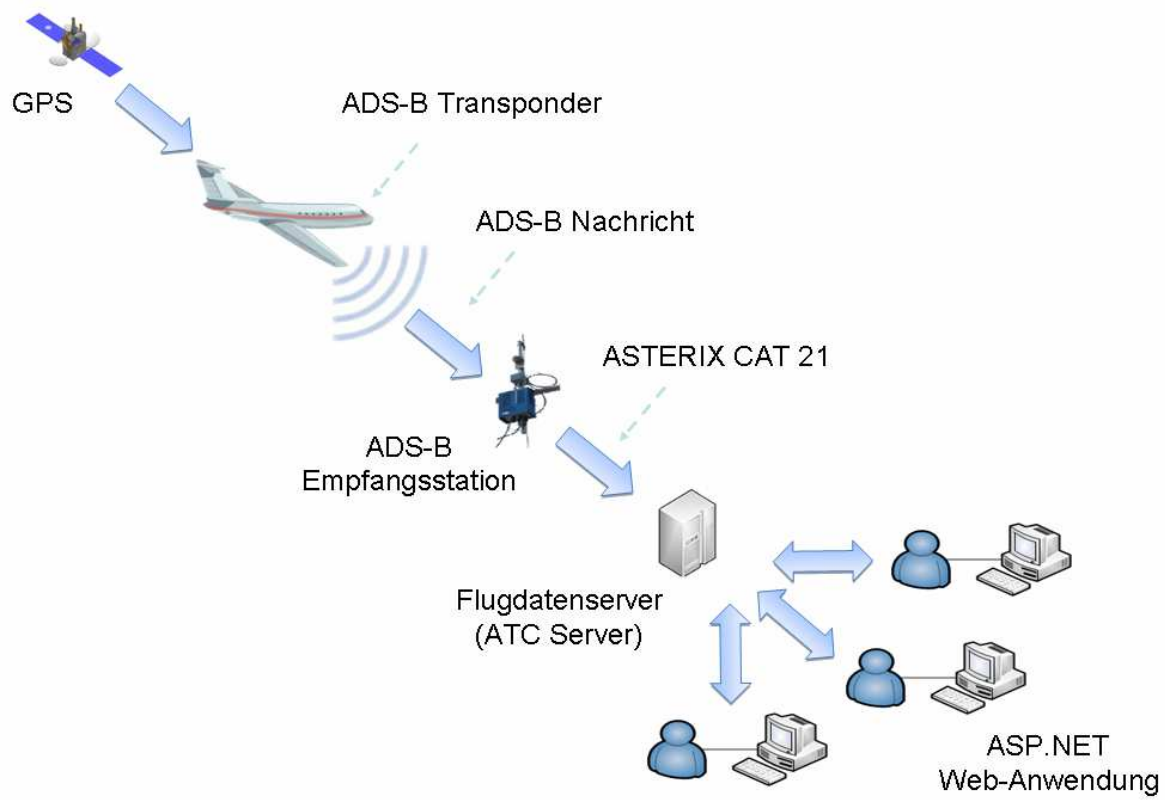


Abb. 2.2.1.1: Informationsfluss zum ADS-B Empfang durch den Flugdatenserver.

2.2.2 Architektur des Flugdatenservers

Die Architektur gliedert sich grob in drei Hauptkomponenten. Bestehend aus verschiedenen Datenquellen die genutzt werden, dem eigentlichen Flugdatenserver *(Air Traffic Control Server)* und dem Web-Client bzw. der Web-Anwendung. Abbildung 2.2.2.1 zeigt diese drei Komponenten im Überblick und machte deren Kommunikationskanäle deutlich. Nachfolgend werden diese einzelnen Komponenten genauer beschrieben.

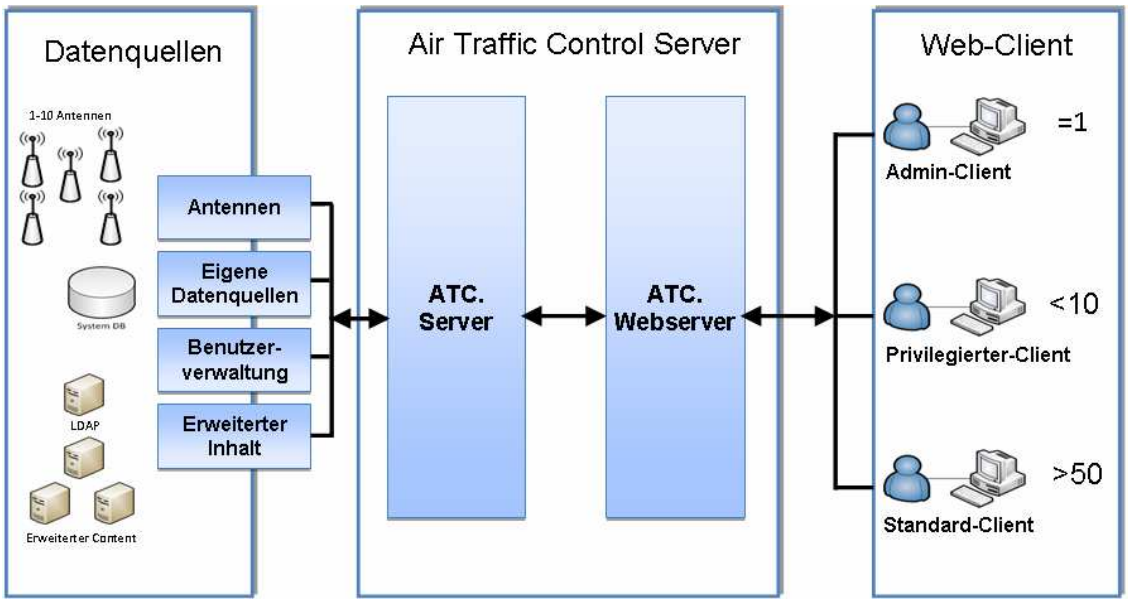


Abb. 2.2.2.1: Architektur bzw. Komponenten des Flugdatenservers.

Datenquellen

Der Flugdatenserver integriert verschiedene Datenquellen über Schnittstellen, die er für seine Funktionalität benötigt.

Die Datenquelle *Antenne* steht für den ASTERIX Cat 21 Nachrichtenempfang von den ADS-B Empfangsstationen (Antennen). Der ASTERIX Cat 21 Nachrichtenempfang wird dabei durch eine UDP-Schnittstelle (User Datagram Protocol) realisiert. Die ADS-B Empfangsstationen senden dabei kontinuierlich ASTERIX Nachrichten an einen oder mehrere definierte UDP-Ports.

Weiterhin werden verschiedene eigene Datenquellen integriert wie z.B. ein Dateizugriff für die Recording/Replay Funktionalität. Mit dieser Funktionalität lassen sich die eingehenden

ASTERIX Cat 21 Nachrichten aufzeichnen und zu einem beliebigen Zeitpunkt wieder abspielen. Im Prinzip können damit Flugbewegungen festgehalten und jederzeit wieder visualisiert werden. Eine weitere eigene Datenquelle stellt die XML-Konfigurationsdatei dar, über die der Flugdatenserver konfiguriert wird.

Für die Authentifizierung der Nutzer wird der LDAP Server der Hochschule Offenburg über eine weitere Schnittstelle verwendet. Dadurch ist es für Studenten der Hochschule Offenburg möglich, den Web-Client des Flugdatenservers mit Ihrem gewöhnlichen Hochschul-Benutzer-Zugang zu nutzen.

Die Datenquelle *Erweiterter Inhalt* bezeichnet eine Schnittstelle zu einem Dienstleister der den Flugdatenserver mit zusätzlichen Flugzeuginformationen versorgen kann. Wie bereits beschrieben, liefert das ASTERIX Cat 21 Format nur Basisinformationen zu einem Flugzeug. Um weitere Flugzeuginformationen zu erhalten, wie z.B. den Ab- und Zielflughafen, die Abflug- und Ankunftszeit etc. wird eine externe Dienstleistung über diese Schnittstelle verwendet. Hier erfolgt die Kommunikation über eine einfache HTTP-Schnittstelle.

Flugdatenserver (Air Traffic Control Server)

Grundsätzlich besteht der Flugdatenserver aus zwei getrennten Komponenten bzw. Prozessen. Zum einen aus dem *ATC.Server-Prozess* (Server-Anwendung) und zum anderen aus dem *ATC.Webserver-Prozess* (Web-Anwendung). Bei dem *ATC.Webserver* Prozess handelt es sich eigentlich um den *w3wp* Prozess *des Microsoft Internet Information Service (IIS)* Web-Servers. Dieser Prozess ist verantwortlich für die Verarbeitung der Web-Anwendungen, die sich auf dem Web-Server befinden.

Der *ATC.Server-Prozess* hat grundsätzlich die Aufgabe die ASTERIX Cat 21 Nachrichten zu empfangen, diese zu verarbeiten und der Web-Anwendung auf Anfrage bereitzustellen. Weiterhin ist der *ATC.Server-Prozess* verantwortlich für die Integration und Verwaltung der verschiedenen Datenquellen. Zudem realisiert er die Funktionalität für die Benutzerverwaltung, den Erweiterten Inhalt, das Reporting etc.

Bei dem *ATC.Webserver-Prozess* handelt es sich, wie bereits erwähnt, um den *w3wp Web-Server-Prozess*. Auf dem Web-Server (IIS) befinden sich die Flugdatenserver Web-

Anwendungen mit dem *Admin-, Standard-* und dem *Privilegierten-Web-Client*. Zudem stellt der Web-Server für jeden *dieser Web-Clients einen separaten Web-Service bereit.*

Durch die Aufteilung des Flugdatenservers in zwei getrennte Prozesse, besteht die Möglichkeit beide Prozesse auf unterschiedlichen Systemen auszuführen. Derzeit werden beide Prozesse aber auf demselben Server-System ausgeführt. Durch die Aufteilung des Flugdatenservers in zwei separate Prozesse, ist ein IPC-Mechanismus (Inter-Prozess-Kommunikation) notwendig, der die beiden kollaborierenden Prozesse wieder miteinander verknüpft (siehe Kapitel 2.2.2).

Web-Client

Grundsätzlich ist zwischen drei verschiedenen Web-Clients bzw. Web-Anwendungen zu unterscheiden. Wie schon erwähnt stellt der *ATC.Webserver* einen *Admin-Web-Client*, einen *Standard-Web-Client* und einen *Privilegierten-Web-Client* zur Verfügung. Diese drei Web-Client-Typen unterscheiden sich in Ihren Rechten und somit in Ihrem Zugriff auf den Flugdatenserver.

Der *Admin-Web-Client* dient zur Browser-basierten Konfiguration des Flugdatenservers. So können z.B. Antennen (Datenquelle Antennen), nach Bedarf hinzugefügt oder auch wieder gelöscht werden. Der *Admin-Web-Client* ist nur für eine geringe Anzahl an Nutzern vorgesehen, typischerweise für den Administrator des Flugdatenservers.

Der *Standard-* und der *Privilegierte-Web-Client* dagegen sind verantwortlich für die Visualisierung der Luftraumbewegungen. Abhängig von der Anmeldung bzw. Authentifizierung erhält der Benutzer Zugriff auf den *Standard-* bzw. *Privilegierten-Web-Client*.

Die Visualisierung der Flugzeugnachrichten erfolgt unter Zuhilfenahme der von Google bereitgestellten Programmierschnittstelle für Google-Earth. Google-Earth bietet ein fertiges 3D-Kartenmateriall, das einfach in die eigene Web-Anwendung integriert und über die eben genannte Schnittstelle angesteuert und verwendet werden kann. Die Flugzeugnachrichten erhält der Web-Client über zeitgesteuerte zyklische Anfragen an einen Web-Service, der sich auf dem *ATC.Webserver* befindet. Die Anfragen erfolgen in einem vorgegeben Intervall (1-30 Sekunden) und werden im Hintergrund der Web-Anwendung mit Hilfe von JavaScript und AJAX (Asynchronous JavaScript and XML) durchgeführt. Mit jeder Anfrage erhält der Web-Client in

der Regel neue Flugzeugnachrichten. Diese werden nach Erhalt unmittelbar ausgewertet und auf der Google-Earth Karte platziert und visualisiert.

Der *Privilegierte-Web-Client* hat im Vergleich zum *Standard-Web-Client* mehr Funktionalität, wie z.B. der Möglichkeit Analysen von Flugbewegungen vorzunehmen. Der wichtigste Unterschied liegt jedoch in der Datenbereitstellung neuer Flugzeugnachrichten. Anfragen über neue Flugzeugnachrichten, die der *Standard-Web-Client* an den *ATC.Webserver* stellt, unterliegen einem sogenannten Delay (zeitliche Verzögerung). Dies bedeutet, dass ein *Standard-Web-Client* nur Flugzeugnachrichten erhält, die älter sind als die definierte Delay-Einstellung. Das Delay ist auf dem *ATC.Server-Prozess* konfiguriert und liegt zwischen 1 bis 60 Minuten. Der *Privilegierte-Web-Client* dagegen unterliegt keinem Delay.

Die beiden Web-Clients (*Standard- und Privilegierter-Web-Client*) bieten insgesamt drei verschiedene Ansichten. Die *Standard-*, *Cockpit-* und die *Shadow-Ansicht*. Bei der *Standard-Ansicht* werden alle verfügbaren Flugzeuge auf dem 3D-Kartenmaterial dargestellt. Bei den beiden anderen Ansichten handelt es sich um eine individuelle Ansicht eines ausgewählten Flugzeuges. Grundsätzlich kann jedes dargestellte Flugzeug auf dem 3D-Kartenmaterial ausgewählt werden. In der *Cockpit-Ansicht* wird die Bewegung des Flugzeuges aus der Cockpit-Perspektive des ausgewählten Flugzeuges dargestellt. In der *Shadow-Ansicht* wird die Bewegung des Flugzeugschattens aus der Vogelperspektive eines Flugzeuges dargestellt. In der nachfolgenden Abbildung 2.2.2.2 wird die *Standard-Ansicht* dargestellt.

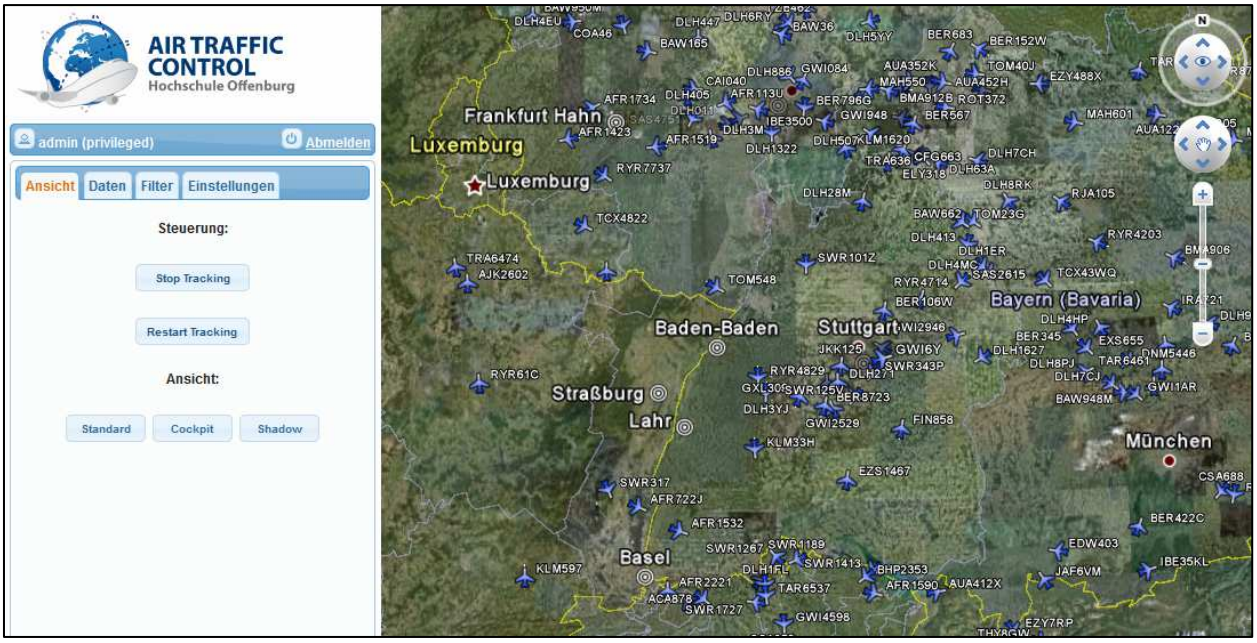


Abb. 2.2.2.2: Standard-Ansicht der Flugdatenserver Web-Anwendung.

2.2.3 Kommunikationsschnittstellen

Grundsätzlich wird die Web-Anwendung auf dem Flugdatenserver (*Admin-, Standard- und Privilegierter-Web-Client*) über das HTTP-Protokoll vom *ATC.Webserver* heruntergeladen und im Browser dargestellt. Die kontinuierliche Versorgung der Web-Anwendung mit neuen Flugzeugnachrichten erfolgt jedoch über die nachfolgenden Kommunikationsschnittstellen:

Kommunikationsschnittstelle	**Beschreibung**
Web-Service	Client/Server Kommunikation zwischen Web-Client und *ATC.Webserver*
TcpChannel	Server-seitige Kommunikation zwischen *ATC.Webserver* und *ATC.Server*

Abb. 2.2.3.1: Kommunikationsschnittstellen des Flugdatenservers

Web-Service

Der *ATC.Webserver* stellt für jeden der in Abschnitt 2.2.2 beschriebenen Web-Clients einen separaten Web-Service zur Verfügung. Somit gibt es einen *Admin-*, einen *Standard-* und einen *Privilegierten-Web-Service*. Der *Standard-* und der *Privilegierte-Web-Service* werden von dem jeweiligen Web-Client genutzt, um kontinuierlich neue Flugzeugnachrichten (Airplane-Objekte) vom *ATC.Webserver* anzufordern, um damit die Visualisierung der Flugbewegungen zu aktualisieren.

Grundsätzlich kann der Web-Service über zwei unterschiedliche Ansätze genutzt werden. Zum einen über das *SOAP-Protokoll* oder direkt über das *HTTP-Protokoll* (*REST-Ansatz*). Der mengenmäßige Overhead, den das *HTTP-Protokoll* erzeugt, ist dabei geringer als über das *SOAP-Protokoll*. Des Weiteren können die beiden Web-Service (Standard- und Privilegierter-Web-Service) die bereitgestellten Flugzeugnachrichten in den Datenformaten XML (Extensible Markup Language) oder JSON (JavaScript Object Notation) bereitstellen. Das JSON-Format wird meist für die Übertragung von JavaScript-Objekt verwendet und ist im Vergleich zum XML-Format einfacher aufgebaut und erzeugt dadurch weitaus weniger Datenvolumen. In den beiden nachfolgenden Listings 2.2.3.1 und 2.2.3.2 erfolgt ein Vergleich der beiden Datenformate anhand einer Flugzeugnachricht.

Im Listing 2.2.3.1 wird eine Flugzeugnachricht im JSON-Format dargestellt. Hier ist der Overhead des Datenformates weitaus geringer, als im Listing 2.2.3.2 mit XML-Format. Der Unterschied der beiden Formate liegt in der Trennung der Attribute. Im XML-Format werden Attribute und deren Werte mit öffnenden und schließenden *Tags* abgebildet. Im JSON-Format werden die Attribute mit einem Komma separiert und Ihre Werte mit einem Doppelpunkt angehängt. Im XML-Format kommt zusätzlich noch der XML-Header hinzu.

```
{
  airplane:{
  seqno:500,
  timestamp:'2010-11-02T16:09:52.7274331+01:00',
  address:4218896,
  callsign:8807,
  altitude:10668,
  longitude:8.407063,
  latitude:49.905674,
  heading:116.809,
  groundspeed:911.53125,
  source:1
  }
}
```

Listing 2.2.3.1: Flugzeugnachricht im JSON-Format (Größe: 250 Bytes).

```
<?xml version="1.0"?>
<ArrayOfAirplane xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Airplane>
    <SeqNo>0</SeqNo>
    <Timestamp>2010-11-02T16:09:52.719933+01:00</Timestamp>
    <Address>4218593</Address>
    <Callsign>BAW641</Callsign>
    <Altitude>11574.78</Altitude>
    <Longitude>8.595498</Longitude>
    <Latitude>48.95666</Latitude>
    <Heading>307.791</Heading>
    <GroundSpeed>783.347168</GroundSpeed>
    <Source>1</Source>
  </Airplane>
</ArrayOfAirplane>
```

Listing 2.2.3.2: Flugzeugnachricht im XML-Format (Größe: 544 Bytes).

Der Web-Client nutzt den Web-Service direkt über das *HTTP-Protokoll (REST-Ansatz)* in Verbindung mit dem JSON-Format. Die Flugzeugnachrichten werden somit in einer Web-Service-Antwort im JSON-Format im *HTTP-Protokoll* „verpackt“ und an den Web-Client übertragen.

TcpChannel

Bei dem *TcpChannel* handelt es sich um einen .NET RPC-Mechanismus (*Remote Procedure Call*), mit dem Prozeduren aus entfernten Prozessen aufgerufen werden können.

Wie zuvor in Abschnitt 2.2.2 beschrieben, werden die Flugzeugnachrichten im *ATC.Server-Prozess* gesammelt. Damit der *ATC.Webserver* bzw. der Web-Service die Web-Clients mit Flugzeugnachrichten versorgen kann, erfolgt auf eine Web-Service-Anfrage ein Server interner RPC-Auruf vom *ATC.Webserver* zum *ATC.Server*. Der *ATC.Server* beantwortet diesen RPC mit einer bestimmten Menge an Flugzeugnachrichten an den *ATC.Webserver*. Der *ATC.Webserver* wiederum „verpackt“ diese erhaltene Menge an Flugzeugnachrichten, wie in diesem Kapitel bereits beschrieben und sendet diese mit der Web-Service-Antwort (Response) an den Web-Client zurück. Die Flugzeugnachrichten werden zwischen dem *ATC.Server* und dem *ATC.Webserver* in einem Binärdatenformat übertragen.

Die Kommunikationsschnittstelle zwischen *ATC.Webserver* und *ATC.Server* ist aufgrund der Aufteilung des Flugdatenservers in getrennte Prozesse notwendig (Interprozess-kommunikation). Dies ermöglicht allerdings die Ausführung der beiden Prozesse auf unterschiedlichen Systemen. In der nachfolgenden Abbildung 2.2.3.1 werden die bidirektionalen Kommunikationskanäle zwischen *ATC.Webserver* und *ATC.Server* dargestellt. Im nachfolgenden Abschnitt wird auf den Ablauf der Kommunikation selber genauer eingegangen.

Abb. 2.2.3.1: Kommunikationskanäle zwischen den Komponenten des Flugdatenservers.

2.2.4 Ablauf der Kommunikation

Nachfolgend wird der Ablauf zur Aktualisierung der Visualisierung bzw. der Kommunikation mit Hilfe von Pseudocode genauer beschrieben. Das Verstehen des folgenden Pseudocodes ist für die spätere Auswertung der Analyseergebnisse aus Abschnitt 4.4 grundlegend.

Grundsätzlich ist zu erwähnen, dass der *ATC.Server* die Flugzeugnachrichten in einem Ringpuffer (Speicherstruktur) vorrätig hält. Jede Flugzeugnachricht ist dabei zusätzlich mit einer eindeutigen Sequenznummer (Folgenummer) und einem Zeitstempel versehen. Die Sequenznummer entspricht dabei der Position im Ringpuffer (Indexposition). Bei dem Zeitstempel handelt es sich um die Eingangszeit der Flugzeugnachricht in die *ATC.Server-Komponente*. Beide Attribute spielen bei der Bereitstellung der Flugzeugnachrichten an die Web-Clients eine große Rolle.

```
1  Standard- und Privilegierter-Web-Client
2
3  seqNo = -1;
4  antenna[] = { 1 };
5
6  while(client is RUNNING)
7  {
8     if Std-Web-Client
9        List[Airplane] a = StdWebService.getAirplane(seqNo);
10    else
11       List[Airplane] a = PrivWebService.getAirplane(seqNo, antenna, “”);
12
13    aktualisiereVisualisierung(a);
14    seqNo = a[a.Count - 1].SeqNo + 1;
15    sleep(Updaterate);
16 }
```

Listing 2.2.4.1: Pseudocode für den Web-Client.

In dem Listing 2.2.4.1 wird die Aktualisierung der Ansicht im Web-Client (*Standard- und Privilegierter-Web-Client*) mit neuen Flugzeugnachrichten (Airplane-Objekten) als Pseudocode dargestellt. Wie einleitend in diesem Kapitel beschrieben, stellen die beiden Web-Clients hiefür kontinuierlich Web-Service-Anfragen. Der *Standard-Web-Client* nutzt dabei den *Standard-Web-Service*, der *Privilegierte-Web-Client* dagegen den *Privilegierten-Web-Service*. Die beiden Web-Service-Aufrufe unterscheiden sich in Übergabe der Parameter.

Beide Web-Service-Methoden erwarten die Übergabe einer Sequenznummer (seqNo). Dabei handelt es sich um dieselbe Sequenznummer, mit dem jede Flugzeugnachricht im Ringpuffer versehen ist. Mit der Angabe einer Sequenznummer informiert der Web-Client den Flugdatenserver darüber, ab welcher Sequenznummer er Flugzeugnachrichten erwartet bzw. welche Flugzeugnachrichten er bereits erhalten hat. Mit der Angabe der Sequenznummer werden Doppelauslieferungen von Flugzeugnachrichten an einen Web-Client ausgeschlossen. In jeder ersten Anfrage eines Web-Clients erfolgt daher die Übergabe der Sequenznummer „-1“. Dies kennzeichnet automatisch die erste Anfrage eine Web-Clients, was zur Folge hat, dass die zu übergebene Menge an Flugzeugnachrichten erst berechnet werden muss. Für die nachfolgenden Anfragen verwendet der Web-Client die jeweils immer um 1 inkrementierte Sequenznummer, der zuletzt erhaltenen Flugzeugnachricht Die Methode des *Privilegierten-Web-Service* verlangt zudem zwei weitere Übergabeparameter. Zum einen eine Liste mit Antennen-Ids (antenna) und zum anderen das *Call Sign* („<callsign>“) des aktuell in der Web-Anwendung ausgewählten Flugzeuges.

Antennen ID

Mit der Übergabe einer Liste von Antennen-Ids, hat der *Privilegierte-Web-Client* die Möglichkeit Flugzeugnachrichten selektiv nur von bestimmten Antennen zu erhalten. Die Antennen die zur Verfügung stehen, können im *Privilegierten-Web-Client* ausgewählt werden.

Call Sign

Das *Call Sign* des ausgewählten Flugzeuges wird nur im Falle der Cockpit- oder Shadow Ansicht übergeben. Ansonsten wird eine leere Zeichenkette („“) übergeben. Auf diesen Parameter wird im nachfolgenden Listing 2.2.4.2 noch einmal eingegangen.

Umgehend nach dem Erhalt neuer Flugzeugnachrichten, wird die Ansicht der Flugbewegungen mit den neu erhaltenen Flugzeugnachrichten aktualisiert. Danach erfolgt wie beschrieben, die Ermittlung der Sequenznummer für die nächste Anfrage und die Wartezeit in der Länge der Update-Rate.

```
1  ATC.WebServer
2
3  public class StdWebService
4  {
5        public List<Airplane> getAirplane(seqNo)
6        {
7              return AtcServer.getAirplane(seqNo)
8        }
9  }
10
11
12 public class PrivWebService
13 {
14       public List<Airplane> getAirplane(source, seqNo, selectedAirplane)
15       {
16             List<Airplane> res = AtcServer.getAirplane(seqNo, source);
17             res = reduceAirplanes(res, selctedAirplane)
18             return res;
19       }
20 }
```

Listing 2.2.4.2: Pseudocode für die beiden Web-Services auf dem *ATC.Webserver*.

In dem Listing 2.2.4.2 wird die Implementierung der beiden Web-Service (*Standard- und Privilegierter-Web-Service*) mit Pseudocode beschrieben.

Im *Standard-Web-Service* (*StdWebService*) wird die Client-Anfrage (Request) direkt (unmittelbar) über den *TcpChannel* an den *ATC.Server* weitergeleitet. Das Ergebnis (Liste von Flugzeugnachrichten) aus der weitergeleiteten Anfrage zum *ATC.Server,* wird dabei direkt mit der Web-Service-Antwort (Response) an den Web-Client zurückgegeben.

Im *Privilegierten-Web-Service* (*PrivWebSevice*) wird die Client-Anfrage (Request) ebenfalls an den *ATC.Server* weitergeleitet. Im Gegensatz zum *Standard-Web-Service* wird das Ergebnis (Liste von Flugzeugnachrichten) der weitergeleiteten Anfrage jedoch weiterverarbeitet. Die Weiterverarbeitung stellt eine Art Optimierung da. Die Optimierung wurde im Vorfeld dieser Thesis entwickelt und testweise im *Privilegierten-Web-Service* integriert. Die Optimierung wird durch die Methode *reduceAirplane()* realisiert. Dieser Methode wird die erhaltene Menge an Flugzeugnachrichten übergeben. Die Methode wiederum filtert aus der Menge an Flugzeugnachrichten, Mehrfachnachrichten eines Flugzeuges anhand des eindeutigen *Call Signs* heraus und reduziert somit die zu übertragende Datenmenge. Damit wird erreicht, dass zu einem Flugzeug immer nur noch die aktuellste Flugzeugnachricht weitergegeben wird. Gerade bei höheren Update-Raten (> 1000 ms) macht sich diese Optimierung bemerkbar, da von einem Flugzeug dementsprechend viele Mehrfachnachrichten vorhanden sind. Wie in Kapitel 2.1.1 beschrieben, versendet ein Flugzeug im Intervall von einer Sekunde seine

Flugzeugnachrichten. Ist beispielsweise eine Update-Rate von 15000 ms definiert, liegen zu jedem Flugzeug nach 15 Sekunden 15 Flugzeugnachrichten vor. Befindet sich der Web-Client jedoch im *Cockpit*- oder in der *Shadow-Ansicht* und wird daher zusätzlich das *Call Sign* des entsprechenden Flugzeuges mit an den *Web-Service* übergeben, wird diese Optimierung für das Flugzeug mit dem entsprechenden *Call Sign* nicht durchgeführt. Für die *Cockpit*- und *Shadow-Ansicht* des ausgewählten Flugzeuges werden diese Mehrfachdaten stets benötigt. Nach der Überprüfung und Reduzierung, wird die neue Liste mit Flugzeugnachrichten an den Web-Client mit der Web-Service-Antwort (Response) zurückgegeben.

```
1  ATC.Server
2
3  public class DataSourceMgmt
4  {
5        public List<Airplane> getAirplane(seqNo)
6        {
7              lastPosition = buffer.getLastPosition();
8
9              If seqNo == -1
10                   seqNo = getPosition(lastPosition, updateRate + delay);
11
12             delayPosition = getPosition(lastPosition, delay);
13
14             return buffer.getAirplane(seqNo, delayPosition);
15       }
16
17       public List<Airplane> getAirplane(seqNo, antenna)
18       {
19             lastPosition = buffer.getLastPosition();
20
21             If seqNo == -1
22                   seqNo = getPosition(lastPosition, updateRate)
23
24             return = buffer.getAirplane(seqNo, lastPosition, antenna)
25       }
26 }
```

Listing 2.2.4.3: Pseudocode für die beiden Methoden zur Datenbereitstellung auf dem *ATC.Server*.

In dem Listing 2.2.4.3 wird ein Ausschnitt zur Implementierung der beiden Methoden zur Datenbereitstellung der *ATC.Server*-Komponente als Pseudocode dargestellt. Die Unterschiede der beiden Methoden sind innerhalb des Listing jeweils explizit rot markiert. Die Methode *getAirplane(seqNo)* wird vom *Standard-Web-Service* genutzt und realisiert die Bereitstellung von Flugzeugnachrichten unter Berücksichtigung des Delays für die *Standard-Web-Clients*. Die Methode *getAirplane(seqNo, antenna)* wiederum wird vom *Privilegierten-Web-Service* genutzt und realisiert die Datenbereitstellung ohne Delay für die *Privilegierten-Web-Clients*.

Grundsätzlich wird in beiden Methoden eine Menge von Flugzeugnachrichten aus einem Bereich der Speicherstruktur (Ringpuffer) ermittelt und an den Web-Service übergeben. Dieser Bereich wird festgelegt durch eine *Startposition* und einer *Endposition*. Als *Startposition* wird die übermittelte Sequenznummer verwendet. Handelt es sich um die erste Anfrage (Sequenznummer = -1) eines Web-Clients, wird die Startposition mit der Methode *getPosition()* berechnet. Die Berechnung erfolgt in den beiden Methoden unterschiedlich. Mit der Methode *getPosition()* kann grundsätzlich die erste Flugzeugnachricht bzw. die Position im Ringpuffer ermittelt werden, deren Zeitstempel unter einer gewissen Zeitspanne liegt. Hierfür erfolgt die Übergabe einer Ausgangspositionen und einer Zeitspanne in Millisekunden. Ausgehend von der Ausgangsposition und deren Zeitstempel, werden nun solange die Zeitstempel der im Ringpuffer dahinter liegenden Flugzeugnachrichten verglichen, bis der Zeitstempel der Flugzeugnachricht unterhalb der übergebenen Zeitspanne liegt. Damit ist die gewünschte Position ermittelt. Als *Endposition* wird in den beiden Methoden eine unterschiedliche verwendet. Für die Datenbereitstellung ohne Delay ist die Endposition die letzte bzw. aktuellste Flugzeugnachricht (lastPosition). Für die Datenbereitstellung mit Delay wird die Endposition bei jedem Aufruf neu berechnet. Hiefür wird ebenfalls die Methode *getPosition()* verwendet. Ist die Startposition und Endposition ermittelt worden, dann werden alle Flugzeugnachrichten aus diesem Bereich des Ringpuffers an den Web-Service zurückgegeben.

Zusammenfassend ist festzuhalten, dass die Bereitstellung von Flugzeugnachrichten unter Berücksichtigung des Delay für den *Standard-Web-Client* aufwändiger ist, da bei jedem Aufruf der benötigte Wert für die Endposition durch eine Vergleich von Zeitstempeln erst herausgefunden werden muss. Genau dieser Umstand ist beim *Privilegierten-Web-Client* nicht notwendig. Zudem wird im *Privilegierten-Web-Service* die erhalten Menge an Flugzeugnachrichten, die er vom *ATC.Server* erhalten hat, durch die beschriebene Optimierungsmaßname reduziert, so das an den Web-Client weitaus weniger Flugzeug-nachrichten zurückgegeben werden. Somit gibt es bei der Bereitstellung der Flugzeug-nachrichten durch den Flugdatenserver zwischen den beiden Delay-Einstellungen jeweils einen Unterschied auf *ATC.Server*- und *ATC.Webserver-Ebene*. Inwiefern sich diese Unterschiede auf die Leistung des Flugdatenservers auswirken, wird in Kapitel 4 beschrieben.

2.2.5 Technologie und Plattform

Der Flugdatenserver wurde auf Basis des Microsoft .NET-Frameworks 3.5 entwickelt. Die Server-Anwendung wurde dabei mit der Programmiersprache .NET C# entwickelt. Bei der Web-Anwendung wurde die Technologie ASP.NET von Microsoft, das JavaScript Framework jQuery und die Google-Earth Programmierschnittstelle (API) eingesetzt.

Die beiden Komponenten des Flugdatenservers (*ATC.Server* und *ATC.Webserver*) sind aktuell auf einem Windows 2008 Server System installiert. Dies bedeutet, dass beide Prozesse auf demselben System bzw. Hardware ausgeführt werden. Dies ist durch die Aufteilung des Flugdatenserver in zwei separate Prozesse jedoch nicht zwingend notwendig.

Der Microsoft Windows 2008 Server (32 Bit) selbst ist als virtuelle Maschine auf einem VM Ware Server installiert. Der Microsoft Windows 2008 Server verfügt derzeit über einen Intel Xeon Dual Core Prozessor mit jeweils 2,33 GHz, einen Arbeitsspeicher von 2 GB sowie eine 1000-Mbit Netzwerkkarte. Der Flugdatenserver ist über ein Gigabit-Switch in das Netzwerk der Hochschule integriert.

Bei dem verwendeten Web-Server handelt es sich um den *Microsoft Internet Information Service (IIS) v 6.1.* Weiterhin wird eine Microsoft SQL Server 2008 R2 Datenbank verwendet.

2.3 Leistungstests und Optimierungen

2.3.1 Einführung

Bei den heutigen Verfahren zur Software-Entwicklung ist das Testen ein wesentlicher Bestandteil. So wird in den verschiedenen Entwicklungsphasen auf unterschiedlichen Ebenen (Unit-Tests, Modul-Tests, Intregrations-Tests und System-Tests) getestet. Bei diesen Tests liegt der Fokus jedoch auf der Überprüfung der Funktionalität bzw. der Anforderungen. Sicherlich gibt es auch Anforderungen hinsichtlich der Leistung einer Anwendung, aber eine völlige Klarheit über das Leistungsverhalten ist dadurch nicht gegeben.

Mit Leistungstests kann die Leistung eines Systems oder einer Anwendung exakt bestimmt werden. Leistung hinsichtlich der Ressourcen Verwendung (CPU, Arbeitsspeicher, Datenträger, Netzwerk) aber auch dem Verhalten bei einer bestimmten oder steigenden Benutzerlast. Die Leistungskriterien sind je nach Anwendung unterschiedlich.

Das Ziel von Leistungstests ist es verschiedene Leistungsindikatoren zu sammeln, diese zu analysieren und mögliche Leistungsengpässe zu identifizieren. Auf dieser Basis können dann Optimierungen vorgenommen werden, um diese Leitungsengpässe zu minimieren bzw. beheben.

2.3.2 Leistungsmetriken

Um die Leistung einer Anwendung zu bewerten, gibt es eine Vielzahl von Kennzahlen bzw. Metriken. Anhand dieser Metriken kann die Leistung einer Anwendung bestimmt und Leistungsengpässe identifiziert werden. Darüber hinaus ist es auch durchaus möglich, Hinweise zu erhalten, warum dieser Leistungsengpass entsteht. Oftmals ist es jedoch nötig, dass weitere tiefer gehende Analysen vorgenommen werden müssen, um die Ursachen für einen Leistungs-engpass hinreichend genau zu ermitteln.

Grundsätzlich unterscheidet man zwischen Metriken zur Ressourcenverwendung wie z.B. CPU-, Arbeitsspeicher-, Netzwerkschnittstellen-Auslastung und anwendungsspezifischen Metriken. Metriken zur Ressourcen Verwendung können wiederum unterteilt werden in System-, Plattform- und Netzwerk-Metriken.

Welche Metriken für eine Leistungsanalyse interessant sind, hängt im Wesentlichen vom Typ der Anwendung ab. Für eine Client-Server-Anwendung bzw. eine Web-Anwendung sind Metriken Server-seitig und Client-seitig interessant. Server-seitige Metriken zur Ressourcen-verwendung und Client-seitige anwendungsspezifische Metriken, wie z.B. Antwortzeiten auf Client-Anfragen.

Nachfolgend werden einige Leistungsmetriken aus der Quelle [MEI04] beschrieben, die auch im Rahmen dieser Arbeit verwendet wurden. Bei einigen dieser Metriken handelt es sich um plattformspezifische Leistungsmetriken, die das Microsoft .NET-Framework standardmäßig zur Leistungsmessung zur Verfügung stellt.

Metriken zur Client/Server-Kommunikation

- Antwortzeit bzw. Latenzzeit (Response-Time)
  Hierbei handelt es sich um die Antwortzeit auf eine Client-Anfrage. Diese Zeit beinhaltet den gesamten Zeitraum beginnend mit der Client-Anfrage (Request) bis zum Erhalt der Server-Antwort (Response) bzw. zur Terminierung des Aufrufs auf der Seite des Clients. Die Antwortzeit des Servers auf eine Client-Anfrage wird immer Client-seitig gemessen.

- Durchsatz (Throughput)
  Mit Durchsatz bezeichnet man eine Menge, die innerhalb eines bestimmten Zeitraumes verarbeitet oder übertragen werden kann.
  Diese Metrik ist allerdings verschieden interpretierbar. Aus Sicht des Web-Servers ist der Durchsatz beispielsweise die Anzahl der Client-Anfragen, die er pro Zeiteinheit annehmen und verarbeiten kann. Aus Sicht des Clients, ist dies wiederum die Anzahl der erhaltenen Bytes oder wie in diesem Fall die Anzahl der erhaltenen Flugzeugnachrichten pro Zeiteinheit.

Metriken zur Ressourcen Verwendung / System

- System / Prozessor / Prozessorzeit (%)
  Diese Metrik lieferte die aktuelle Prozessor-Auslastung, verursacht durch alle laufenden Prozesse (Anwendungs- und Betriebsystemprozesse) in Prozent.
  Die Prozessauslastung ist generell ein recht einfacher Indikator zur Bewertung der Leistung. Hält sich die CPU-Auslastung dauerhaft über einem Schwellwert von 85 %, können Leistungsengpässe entstehen.

- System / Prozessor / Kontextwechsel pro Sekunde

  Diese Metrik liefert die Anzahl der Kontextwechsel pro Sekunde. Bei einer hohen Rate ist dies ein Indikator für eine große Anzahl an konkurrierenden in Ausführung befindlichen Threads, was sich dementsprechend auf die CPU-Auslastung auswirkt.

  Kontextwechsel Raten von unter 5000 pro Sekunde liegen im normalen Bereich. Liegen diese darüber ist das ein Indikator für einen Leistungsengpass.

- System / Prozess / Prozessorzeit (%)

  Diese Metrik lieferte die verwendete Prozessorzeit eines Prozesses der sich in Ausführung befindet in Prozent. Damit kann der Zeitanteil an der aktuellen CPU-Auslastung eines bestimmten Prozesses bestimmt werden.

  Die Prozessorzeit ist ebenfalls ein wichtiger Indikator, da hiermit einem Prozess direkt CPU-Nutzung (Zeit) zugeordnet werden kann.

- System / Speicher / Verfügbare MB

  Diese Metrik liefert die Menge des physikalischen Arbeitsspeichers in Mega Bytes (MB), die auf dem System aktuell noch zur Verfügung steht.

  Hiermit lässt sich ein Leistungsengpass im Arbeitsspeicher bestimmen.

- System / Physikalischer Datenträger / Durchschnittliche Warteschlangenlänge

  Diese Metrik liefert die durchschnittliche Warteschlangenlänge für alle Schreib- und Leseoperationen auf einen physikalischen Datenträger.

  Ist diese Rate zu Hoch, ist das ein Hinweis darauf, dass das System mit den Datenträgeroperationen nicht mehr nachkommt.

- System / Netzwerk / Bytes pro Sekunde

  Diese Metrik liefert die Gesamtanzahl an Bytes pro Sekunde, die über eine bestimmte Netzwerkschnittstelle versendet und empfangen wurden.

  Die Bytes pro Sekunde geben Aufschluss darüber ob eine bestimmte Netzwerkschnittstelle für ein bestimmtes Datenvolumen ausreichend ist.

Metriken zur Ressourcen Verwendung / Plattform

- .NET CLR / Speicher / #Gen 0 Collections, #Gen1 Collections, #Gen 2 Collections

  Diese Metrik liefert die Anzahl, wie oft der .NET CLR Garbage Collector für Gen 0, Gen 1 und Gen 2 Objekte aktiv war.

  Grundsätzlich teilt der .NET CLR Garbage Collector die Objekte einer Anwendung nach Ihrer Lebenszeit in die Generationen 0, 1 und 2 ein. Das Abräumen von Objekten durch den Garbage Collector in der Generation 1 ist deutlich schneller und günstiger als das Abräumen von Objekten der Generation 2. Deshalb sollte hier grundsätzlich ein Verhältnis von 100:10:1 (Gen 0, Gen 1, Gen 2) bestehen. Ist das nicht der Fall, wird die CPU zusätzlich durch den .NET Garbage Collector belastet, verursacht durch die Anwendung.

- .NET CLR / Ausnahmen / Gefangene Ausnahmen pro Sekunde

  Diese Metrik liefert die Anzahl der eingefangenen bzw. verarbeitenden Ausnahmen pro Sekunde einer bestimmten .NET Anwendung.

  Diese Metrik stellt eine einfache Möglichkeit dar, die Anzahl der ausgelösten Ausnahmen innerhalb einer Anwendung zu ermitteln. Dies ist aus dem Grund interessant, da die Verarbeitung von Ausnahmen relativ aufwändig ist und damit die Leistung einer Anwendung beeinflussen kann.

- .NET CLR / Remote / Remote-Aufrufe pro Sekunde

  Diese Metrik liefert die Anzahl der .NET-Remote-Aufrufe pro Sekunde einer bestimmten .NET Anwendung.

  Die Metrik gibt einen Hinweis auf Prozesskommunikation zwischen verschiedenen .NET-Anwendungen über einen .NET-Remote-Mechanismus. Im Prinzip stellt diese Metrik aus Sicht des Servers den Durchsatz an Remote-Aufrufen pro Sekunde auf Prozess-kommunikationsebene dar.

  Im Rahmen dieser Arbeit ist diese Metrik ein wichtiger Indikator um festzustellen, ob der Flugdatenserver eine bestimmte Benutzerlast auf Prozesskommunikationsebene (*ATC.Webserver* und *ATC.Server*) verarbeiten kann.

- ASP.NET / Anfragen pro Sekunde (Requests/sec)

  Diese Metrik liefert die Anzahl der am Web-Server (IIS) eingegangen bzw. verarbeitenden Anfragen (Requests) durch den ASP.NET Worker Prozess (w3wp.exe).

Diese Metrik bestimmt den Durchsatz an verarbeiteten Client-Anfragen pro Zeiteinheit auf Web-Server-Ebene.

Wie bereits bei der zuvor beschriebenen Metrik „Remote-Aufrufe pro Sekunde“, handelt es sich hier wiederum um einen wichtigen Indikator, inwiefern der *ATC.Webserver* die Client-Anfragen verarbeiten kann.

- ASP.NET / Anfragen Ausführungszeit (Request Execution Time)

  Diese Metrik liefert die Zeit die der ASP.NET Worker Prozess zur Ausführung der letzten Anfrage (Request) benötigt hat.

  Auch diese Metrik gibt ein Hinweis auf einen Leistungsengpass. Werden die Ausführungszeiten auf dem Web-Server länger, bedeutet dies, dass der Server eine große Menge an Anfragen verarbeiten muss und sich dadurch die Wartezeiten für die Ausführung einer einzelnen Anfrage der erhöhen.

- ASP.NET / Anfragen in Wartenschlage (Requests in Application Queue)

  Diese Metrik liefert die Anzahl der Anfragen (Requests) die sich in der Warteschlange zur Ausführung durch den ASP.NET Worker Prozess befinden. Diese Kennzahl sollte dauerhaft nicht über 0 (Null) liegen, ansonsten ist dies ein Anzeichen dafür, dass der Web-Server mit der Verarbeitung der Anfragen nicht nachkommt.

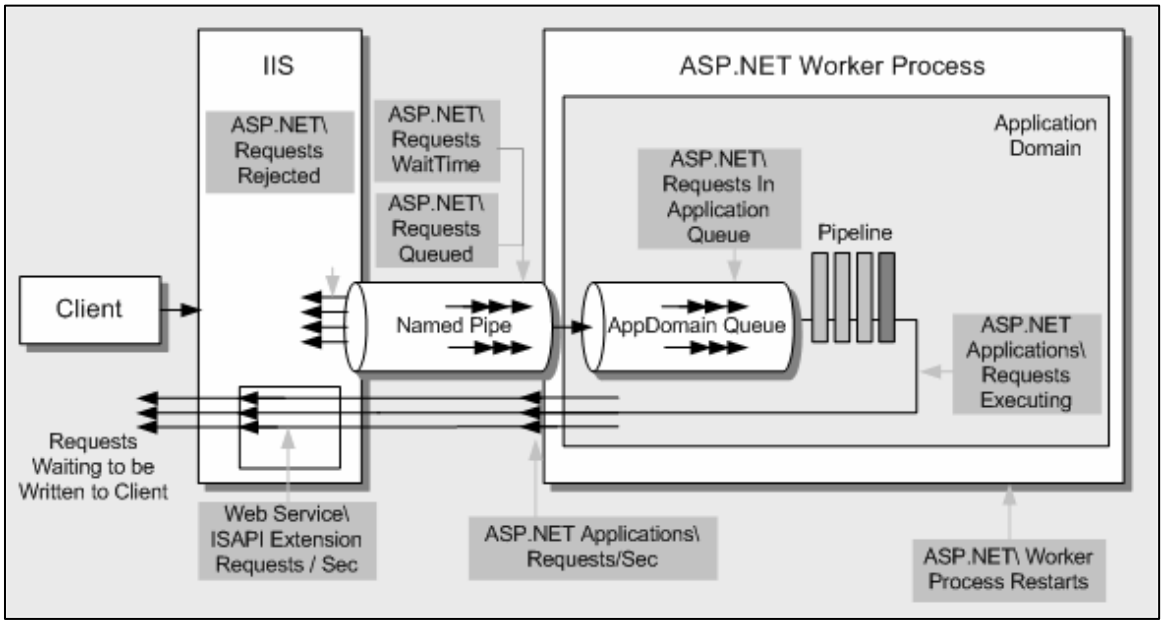


Abb. 2.3.2.1: Leistungsmetriken zur Verarbeitung von Web-Server-Anfragen. [MEI04]

2.3.3 Messung von Leistungsmetriken

Leistungsmetriken können auf verschiedene Weise und mit unterschiedlichen Tools ermittelt werden. Je nach Problemstellung sollte man eine entsprechende Technik auswählen. Nachfolgend werden einige Möglichkeiten beschrieben, wie auf Basis des Betriebssystems Windows und dem .NET-Framework Leistungsmetriken ermittelt bzw. gesammelt werden können.

- Windows Performance Monitor (perfmon.exe)

  Eine einfache Möglichkeit zur Messung von Leistungsmetriken stellt der *Performance Monitor* von Windows dar. Hier können sämtliche Leistungsmetriken, die auf dem System zur Verfügung stehen, ausgewählt und über einen definierten Zeitraum beobachtet und gesammelt werden. Des Weiteren bietet der *Performance Monitor* die Möglichkeit, Diagramme aus den gesammelten Daten zu generieren sowie die gesammelten Messdaten in andere Dateiformate zu exportieren.

- .NET-Framework

  Das .NET-Framework bietet die Möglichkeit, die auf dem System zur Verfügung stehenden Leistungsmetriken über die Klasse *PerformanceCounter* aus dem Namespace *System.Diagnsotics* in einer eigenen Implementierung abzurufen.

  Im Prinzip ist man dadurch in der Lage, abhängig von den eigenen Anforderungen, ein eigenes Tool zur Performance-Messung zu entwickeln. Der Vorteil liegt hier in der Flexibilität, mit der die Leistungsmetriken verwendet werden können.

- Externe Tools

  Es gibt zahlreiche Tools (Profiling Tools, Visual Studio Team System), ähnlich wie der Windows *Performance Monitor*, mit denen man ebenfalls Leistungsmetriken abfragen und Messdaten sammeln kann.

2.3.4 Performance Testing

Das *Performance Testing* beinhaltet Tests, um das Verhalten eines Systems unter einer steigenden bzw. bestimmten Benutzerlast zu bestimmen. Gerade bei Web-Anwendungen, die potenziell für Millionen von Benutzen über das Internet erreichbar sind, ist das Wissen über das Verhalten mit einer steigenden Benutzerlast sehr wichtig.

Eine Web-Anwendung funktioniert bei einer geringen Benutzerlast in der Regel problemlos. Ist aber eine gewisse Schwelle an Benutzerlast erreicht, können schnell Leistungsengpässe entstehen. Ein Leistungsengpass kann sich in Form von langen Antwortzeiten auf Client-Anfragen, bis hin zu einer Nicht-Erreichbarkeit des Web-Servers bemerkbar machen. Mit *Performance Testing* kann das Verhalten bei steigender Benutzerlast, die Schwelle (Peak Load) für die maximale Benutzerlast sowie Leistungsengpässe ermittelt werden.

Für das *Performance Testing* werden in der Regel keine reellen Benutzer für das Testen eingesetzt, sondern Tools die eine bestimmte Menge an Benutzerlast simulieren. In der nachfolgenden Abbildung 2.3.4.1 wird die Funktionsweise eines solchen Tools dargestellt. [MEI04]

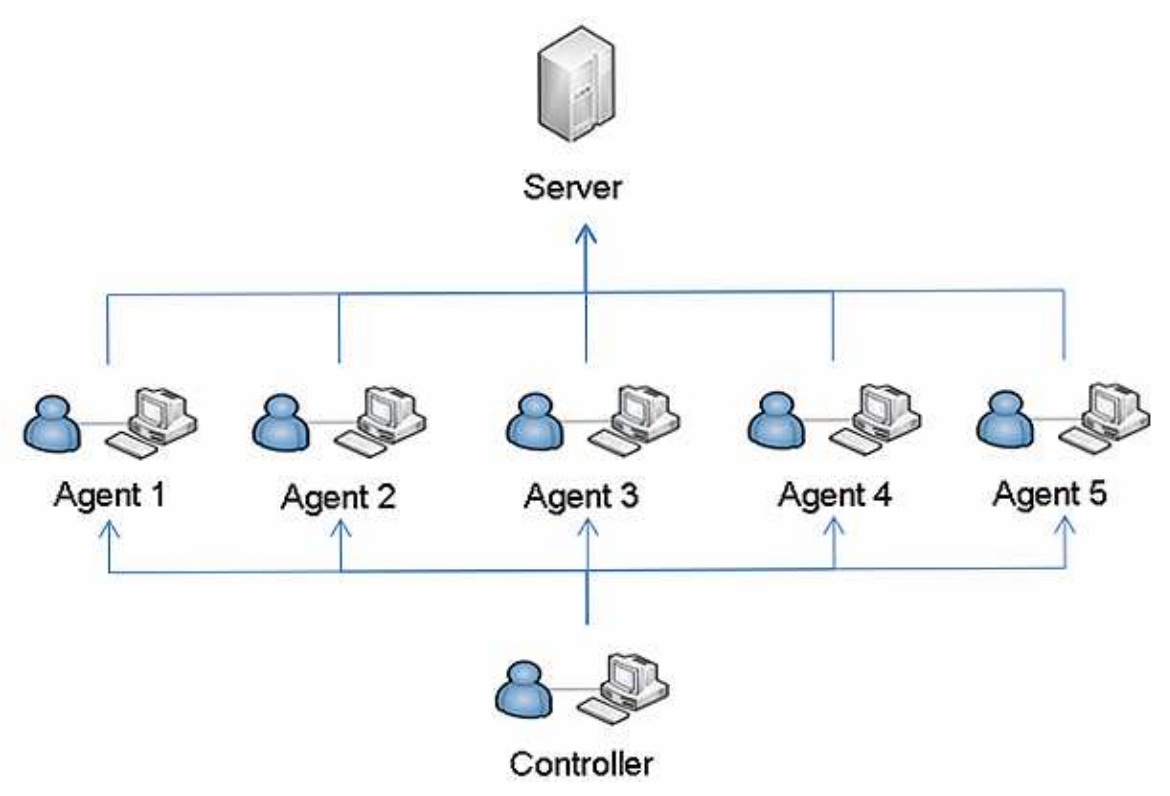


Abb. 2.3.4.1: Funktionsweise eines Performance Testing Tools.

Mit *Performance Testing* alleine lässt sich jedoch noch kein Leistungsengpass bestimmen. Im Prinzip wird nur eine bestimmte Benutzerlast simuliert. Um tatsächlich einen Leistungsengpass unter einer bestimmten Benutzerlast bestimmen zu können, ist das Ermitteln von verschiedenen Leistungsmetriken wie z.B. CPU-, Arbeitsspeicher-Auslastung, Antwortzeiten auf Client-Anfragen und Durchsatz während eines *Performance Tests* notwendig. Anhand der ermittelten Leistungsmetriken, kann das Verhalten bestimmt und Leistungsengpässe erkannt werden.

Mit *Performance Testing* werden unter anderem nachfolgende Zusammenhänge ermittelt [MEI04]:

- Ressourcenverwendung vs. Benutzerlast

Bei einer steigenden Benutzerlast nehmen die Client-Anfragen pro Zeiteinheit zu, die auf dem Server (Web-Server) verarbeitet werden müssen. Für die Verarbeitung der Client-Anfragen werden entsprechend mehr Ressourcen wie z.B. CPU-Zeit benötigt.

Der Anstieg der Ressourcenverwendung sollte verhältnismäßig erfolgen, so dass sich bei einer doppelten Benutzerlast eine doppelte Ressourcenverwendung ergibt. Ist dies der Fall, spricht man von einem skalierbaren System. In der Abbildung 2.3.4.2 ist jeweils ein Beispiel zu sehen für ein proportionales und ein exponentielles Verhältnis. Steigt die Ressourcenverwendung exponentiell, ist das generell ein Hinweis auf einen Leistungsengpass innerhalb der Anwendung.

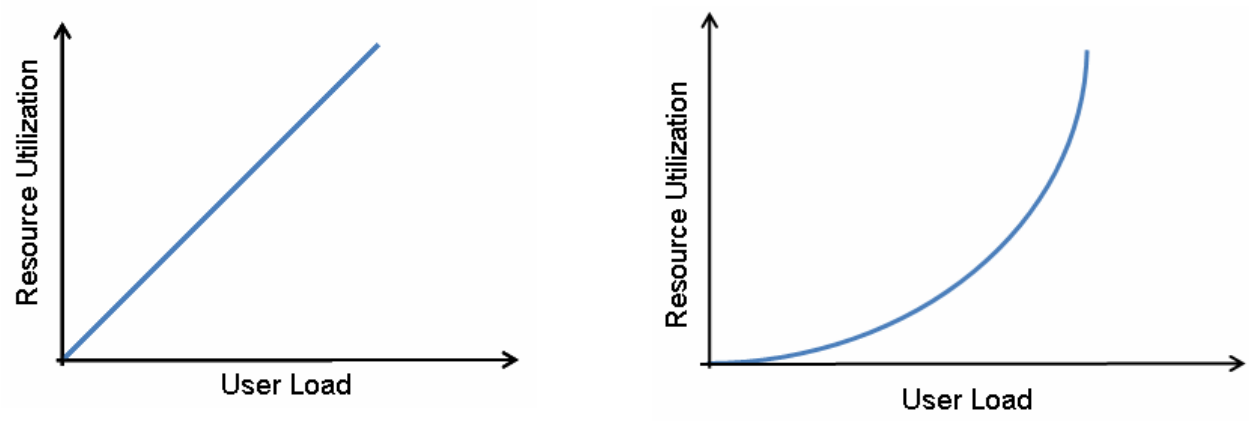


Abb. 2.3.4.2: Proportionaler und Exponentieller Anstieg der Ressourcenverwendung mit einer steigenden Benutzerlast.

- Antwortzeiten und Durchsatz vs. Benutzerlast

Durch eine erhöhte Benutzerlast und der damit verbundenen Ressourcenverwendung, wird für die Verarbeitung der Client-Anfragen mehr Zeit benötigt. Im Prinzip lässt sich das mit mehreren konkurrierenden Threads auf einem System vergleichen. Umso mehr Threads auf einem System aktiv sind, desto länger werden die Wartezeiten der einzelnen Threads, die sich durch das Scheduling des Betriebsystems ergeben.

Mit den eingehenden Client-Anfragen auf dem Server verhält sich dies ähnlich. Umso mehr Client-Anfragen pro Zeiteinheit eingehen, desto länger werden die Wartezeiten bzw. Ausführungszeiten auf dem Server. Die längeren Ausführungszeiten auf dem Server wirken sich dann dementsprechenden auf die Antwortzeiten der Client-Anfragen aus. In Abbildung 2.3.4.3 wird ein typisches Verhalten der Antwortzeiten zur Benutzerlast dargestellt. Bis zu einer bestimmten Schwelle an Benutzerlast (Peak-Load) verhalten sich die Antwortzeiten relativ konstant. Ist diese Schwelle jedoch erreicht, steigen die Antwortzeiten exponentiell.

Durch die längeren Ausführungszeiten zur Verarbeitung der Client-Anfragen wiederum, verringert sich gleichzeitig der Durchsatz an Client-Anfragen pro Zeiteinheit. Dieses Verhalten wird ebenfalls in Abbildung 2.3.4.3 dargestellt (Throughput vs. User Load).

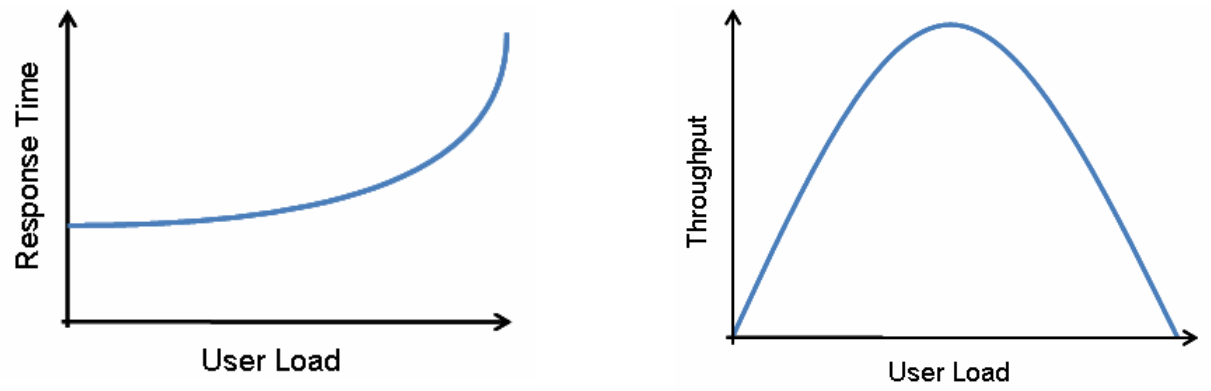


Abb. 2.3.4.3: Typisches Verhalten der Antwortzeiten und dem Durchsatz an Client-Anfragen mit einer steigenden Benutzerlast.

Bis zu einer bestimmten Schwelle an Benutzerlast ist die Verarbeitung der Client-Anfragen effizient im Sinne von akzeptablen Antwortzeiten und Durchsatz. Diese Schwelle (Peak-Load) kann mit *Performance Testing* ermittelt werden.

Mit *Performance Testing* und der Ermittlung von Leistungsmetriken lassen sich nicht nur Leistungsengpässe bestimmen, sondern auch die verantwortlichen Komponenten bzw. Funktionalitäten. Des Weiteren kann gegebenenfalls anhand der Leistungsmetriken auch die Gründe für einen Leistungsengpass ermittelt werden. Dies muss aber nicht immer so sein. In vielen Fällen liefert das *Performance Testing* und die Leistungsmetriken nur Hinweise auf die Gründe für einen Leistungsengpass. Wenn sich beispielsweise die CPU-Auslastung bei annähernd 100% befindet, kann diese viele Ursachen haben.

Grundsätzlich unterscheidet man beim *Performance Testing* zwischen drei verschiedenen Arten an Tests. Dem Capacity *Test*, dem *Load Test* und dem *Stress Test*. [MEI04] [GLAV10]

- Capacity Test

  Beim *Capacity Test* beginnt man mit dem Test einer geringen Benutzerlast. Mit jedem weiteren Testlauf wird die Benutzerlast kontinuierlich erhöht, bis die maximale Benutzerlast bzw. eine Kapazitätsgrenze erreicht ist. Die maximale Benutzerlast kann mit einer Bewertung der Messdaten der Leistungsmetriken, die während einer Benutzerlastsimulation ermittelt werden, bestimmt werden. Mit diesem Test lässt sich das Verhalten eines Systems bei steigender Benutzerlast, der *Peak Load* sowie Leistungsengpässe ermitteln.

- Load Test

  Beim *Load Test* dagegen wird über einen längeren Zeitraum mit einer Benutzerlast getestet, die etwas unterhalb der maximalen Kapazitätsgrenze (*Peak Load*) liegt. Für die maximale Benutzerlast gibt es natürlich keinen allgemein gültigen Wert, vielmehr ist diese Abhängig von der Anwendung. Mit dem *Load Test* wird beobachtet, wie sich das System am *Peak Load* über mehrere Stunden bzw. Tage verhält.

- Stress Test

  Beim *Stress Test* wird ein System an der maximalen Leistungsgrenze (Kapazitätsgrenze) betrieben. Damit wird das Verhalten unter enormer Auslastung ermittelt. Es werden Fragen beantwortet wie:

  - Was passiert mit dem System bei Überlastung?
  - Gehen Client-Anfragen verloren?
  - Reagiert das System überhaupt noch?

Performance Testing Prozess

Wie bereits ausführlich beschrieben geht es beim *Performance Testing* darum, das Verhalten des Systems in Abhängigkeit zur Benutzerlast zu bestimmen. Durch die Messung verschiedener Leistungsmetriken wie CPU-, Arbeitsspeicher-Auslastung, Antwortzeiten auf Client-Anfragen etc. erhält man somit eine Menge an Messreihen, die in Abhängigkeit mit der Benutzerlast stehen. Auf Basis dieser Messreihen kann die Leistung des Systems dann bewertet werden.

Um *Performance Testing* strukturiert durchzuführen, bietet sich die Verwendung des *Performance Testing* Prozess an, der in der Quelle [MEI04] beschrieben wird.

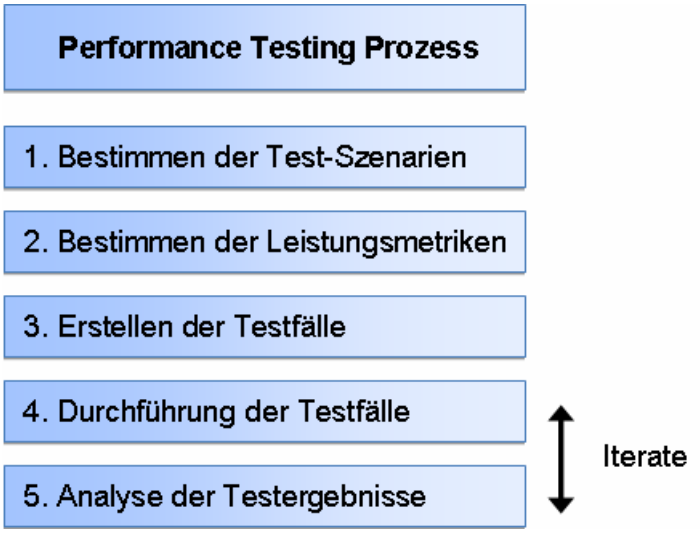


Abb. 2.3.4.4: *Performance Testing* Prozess

1. Bestimmen der Testszenarien

Der erste Schritt ist die Identifizierung der verschiedenen relevanten Testszenarien, die getestet werden sollen. Ein Testszenario ist beispielsweise eine bestimmte Funktionalität innerhalb einer Anwendung. Im Hinblick auf eine Web-Anwendung kann ein Testszenario auch eine Folge von Client-Anfragen (Funktionszweig) wie z.B. 1. Login, 2. Suche nach einem Produkt, 3. Kauf des Produktes sein. Bezogen auf die Web-Anwendung des Flugdatenservers, kann hier beispielsweise der Aufruf des *Standard-Web-Service* zur Aktualisierung der Visualisierung der Flugbewegungen als Testszenario genannt werden. Für die Erstellung der verschiedenen

Testszenarien ist es wichtig, dass man über das Benutzerverhalten innerhalb der zu testenden Anwendung informiert ist. Zu mindestens sollte man aber wissen, welche Funktionalität besonders wichtig ist. Besonders hilfreich zur Bestimmung von relevanten Testszenarien können Use Case UML-Diagramme sein. Hiermit lassen sich wichtige Testszenarien relativ einfach ableiten. Auch so genannte Site Maps, die das Benutzerverhalten auf einer Webseite dokumentieren, können als Informationsquelle für Testszenarien herangezogen werden.

2. Bestimmen der Leistungsmetriken

Im zweiten Schritt erfolgt die Bestimmung der verschiedenen Leistungsmetriken (z.B. CPU-Auslastung, Antwortzeiten auf Client-Anfragen, Durchsatz), die während der Testläufe der Testszenarien gemessen werden. Für jedes Testszenario können theoretisch unterschiedliche Leistungsmetriken verwendet werden.

3. Erstellen der Testfälle

Auf Basis der definierten Testszenarien werden im dritten Schritt Testfälle definiert. Ein Testszenario kann beispielsweise unter verschiedenen Konfigurationen oder einer bestimmten Benutzerlast ausgeführt werden. Im Prinzip wird das Testszenario noch einmal in beliebig viele Testfälle herunter gebrochen.

4. Durchführung der Testfälle

Im vierten Schritt erfolgt die Ausführung der definierten Testfälle. Dies beinhaltet die Simulation der Benutzerlast und das Messen bzw. Sammeln der Messwerte der verwendeten Leistungsmetriken. Für jeden Testfall liegen nach der Ausführung Ergebnisse über die Messung der im Test verwendeten Leistungsmetriken vor.

5. Analyse der Testergebnisse

Im fünften und letzten Schritt werden die erhaltenen Messreihen analysiert. Die erhaltenen Messergebnisse dienen zudem als Grundlage (Baseline) für den Vergleich mit den Messwerten aus früheren oder späteren Testläufen z.B. nach der Umsetzung von Optimierungen.

Die Schritte 4 und 5 können nach der Durchführung von Optimierungen iterative ausgeführt werden.

2.3.5 Application Profiling

Mit dem oben beschriebenen *Performance Testing* lässt sich das Verhalten einer Anwendung bei einer bestimmten Benutzerlast beobachten. Leistungsengpässe und die maximale Benutzerlast eines Systems können ebenfalls ermittelt werden. Doch die Ursachen für das Verhalten oder für den Leistungsengpass können damit in der Regel nicht ermittelt werden.

Mit *Application Profilern* können Anwendungen tiefer gehend analysiert werden. *Application Profiler* sind in der Lage zur Laufzeit einer Anwendung Funktions- und Speichernutzung zu analysieren.

Grundsätzlich unterscheidet man zwischen *Performance Profiling* und *Memory Profiling*. Beim *Performance Profiling* wird das Verhalten von Funktionen in der Anwendung analysiert. Das *Memory Profiling* dagegen analysiert die Speicherverwendung einer Anwendung. [GLAV10]

Performance Profiling

Performance Profiler sind in der Lage das Verhalten von Funktionen einer Anwendung zur Laufzeit zu analysieren. Nach der Quelle [GLAV10] können *Performance Profiler* dabei folgende Leistungsmetriken einer Funktion analysieren:

- Elapsed Time / Wall Clock Time
  Die *Elapsed Time* entspricht der Laufzeit einer Funktion. Allerdings enthält diese die tatsächlich benötigte Prozessorzeit inklusive der Wartezeiten, die sich durch die Verdrängung durch andere Prozesse (Scheduling) ergeben und nicht nur die tatsächlich verwendete Prozessorzeit.

- CPU / Application Time
  Tatsächlich verwendete Prozessorzeit ohne Wartezeit.

- Hit Count
  Anzahl der Aufrufe einer Funktion.

- Network / Disk Activity
  Anzahl der Bytes für Netzwerk- oder Datenträger Operationen einer bestimmten Funktion.

Für sämtliche der oben genannten Metriken werden zudem MIN (Minima), AVG (Durchschnitt) und MAX (Maxima) Werte ermittelt. Darüber hinaus bieten *Performance Profiler* die nachfolgende Funktionalität:

- Call tree
  Ermittlung eines Call Graphs, der die Beziehungen zwischen Funktionen bzw. von Funktionsaufrufen darstellt.

- Line based
  Zeilenbasierte Laufzeitanalyse (CPU / Application Time) einer Funktion. Ein *Profling Tool* ermöglicht damit eine einfache Möglichkeit CPU-, Netzwerk- oder Datenträger-intensive Funktionen zu ermitteln und damit auch eine mögliche Ursache für einen Leistungsengpass.

Profiling Tools ermöglichen einen sehr tiefen Einblick in das Verhalten einer Anwendung. Je nach Profiler-Typ entsteht jedoch ein Leistungs-Overhead durch die Ermittlung (Profiling) der verschiedenen Metriken. Im Grunde gibt es drei Hauptarten von Profilern, die sich in der Vorgehensweise zur Messwertermittlung im Leistungs-Overhead unterscheiden:

- Instrumenting Profiler
  Instrumenting Profiler liefern sehr exakte Ergebnisse indem Sie die Anwendung mit eigenem Code instrumentieren bevor Sie mit dem *Profiling* beginnen. Instrumentieren bedeutet, dass der bestehende Programmcode um weiteren Code, der zur Messung dient, erweitert wird. So wird z.B. vor und nach jedem Funktionsaufruf der instrumentierte Code aufgerufen, um die Laufzeit der Methode zu ermitteln.
  Instrumenting Profiler haben den Nachteil, dass Sie einen entsprechenden Leistungs-Overhead erzeugen und das Programm nicht mehr ohne Leistungsverlust ausgeführt werden kann (Leistungseinbußen). Auf der anderen Seite liefern Sie hinreichend genaue Ergebnisse.

- Sampling profiler
  Sampling Profiler dagegen arbeiten mit Stichproben und einem statistischen Ansatz. Hierfür überprüft der *Sampling Profiler* zyklisch den aktuellen Befehlszähler (Program Counter) und kann daraus statistisch die Anzahl der Aufrufe und Aufrufbeziehungen von Funktionen einer

Anwendung ableiten. Da es sich um einen statistischen Ansatz handelt, sind die Messergebnisse allerdings nicht exakt.
Der Vorteil dabei ist, dass die Anwendung selbst nicht verändert wird und damit ohne Leistungsverlust ausgeführt werden kann.

- Event Based profiler
  Laufzeitumgebung wie die .NET CLR oder die Java Virtual Machine können so konfiguriert werden, dass Sie bei bestimmten Events (Speicher Allokierung, Funktionsaufrufen, Exceptions etc.) Nachrichten an einen Profiler melden. Anhand dieser Nachrichten kann der Profiler dann seine Analysen vornehmen.
  Auch bei diesem Verfahren entsteht ein Leistungs-Overhead der sich jedoch weniger auf die zu testende Anwendung sondern mehr auf das System auswirkt.

Einen bestimmten *Application Profiler* Typ für alle Leistungsanalysen zu benutzen ist daher nicht möglich, vielmehr ist dieser abhängig von der Anwendung und dem Ziel der Leistungsanalyse. In einigen Analysefällen können durchaus *Sampling Profiler* völlig ausreichen.

Memory Profiling

Memory Profiler analysieren die Speicherverwendung einer Anwendung zur Laufzeit und können dabei typische Speicherprobleme identifizieren. Nachfolgend werden einige typische Probleme die bei der Speicherverwendung auftreten können beschrieben. [GLAV10]

- Exzessives Allokieren von temporären Objekten
  Hierbei handelt es sich um das Allokieren einer Vielzahl von temporären Objekten die nicht benötigt werden. Dadurch werden unnötigerweise zusätzliche System-Ressourcen benötigt, um diese nicht benötigten Objekte wieder aus dem Speicher zu löschen.

  Ein typisches Beispiel für eine unnötige Allokierung ist die Verkettung von Zeichenketten in .NET. Mit der Verkettung von Zeichenketten nach dem Beispiel in Listing 2.3.5.1 werden 1001 Objekte im Speicher allokiert. Im Prinzip bleibt aber nur eine Referenz auf ein Objekt bestehen. Die anderen 1000 Objekte werden vom Garbage Collector abgeräumt, was entsprechend System-Ressourcen in Anspruch nimmt.

```
1  String text = “Hallo Welt“;
2
3  for (int i=0; i<1000,i++)
4  {
5        text += “Hallo Welt”
6  }
```

Listing 2.3.5.1: Beispiel für ein unnötiges Allokieren von temporären Objekten.

Effizienter funktioniert die Zeichenketten Verkettung mit Hilfe eines sogenannten *StringBuilders*. Hier wird tatsächlich nur ein Objekt erzeugt.

- Mid-life crisis

  Dieses Problem steht im Zusammenhang mit dem .NET-Framework und dem dortigen Garbage Collector Mechanismus.

  Wie bereits beschrieben, teilt der .NET CLR Garbage Collector die Objekte einer Anwendung nach Ihrer Lebenszeit in die Generationen 0, 1 und 2 ein. Ein Grund dafür ist unter anderem einen effizienterer Garbage Collector Mechanismus.
  Das Abräumen bzw. Löschen von Objekten durch den Garbage Collector der 1. Generation ist deutlich schneller und günstiger als das Löschen der Objekte der 2. Generation. Tritt jedoch der Fall ein, dass der Garbage Collector sehr häufig für das Abräumen von Objekten der 2. Generation aktiv werden muss, wird die CPU zusätzlich durch den Garbage Collector belastet.

- Memory Leaks

  Bei *Memory Leaks* handelt es sich um Speicher der nicht mehr genutzt wird aber nicht freigegeben wurde.
  In Programmiersprachen wie C/C++ muss allokierter Speicher durch den Entwickler explizit wieder freigegeben werden. In Sprachen ohne Garbage Collector ist die Gefahr von *Memory Leaks* deshalb besonders groß. Aber selbst beim Managed Code in Laufzeitumgebungen wie der JRE oder der .NET, die einen Garbage Collector Mechanismus besitzen, können *Memory Leaks* entstehen, indem Objektreferenzen nicht gelöscht werden, obwohl diese nicht mehr benötigt werden.

Aktuelle Memory Profiler bieten die nachfolgende Funktionalität zur Analyse der Speicherverwendung:

- Memory Snapshot
  Ein *Memory Snapshot* ist ein Speicherabbild der Anwendung zur Laufzeit. Dieses Abbild kann für eine Analyse zur Arbeitsspeicherverwendung, explizit zur Heap-Verwendung oder zur Analyse der Anzahl allokierter Objekte verwendet werden.

- Identifizierung ineffiziente Speichernutzung

- Die oben beschriebenen Speicherprobleme (Exzessives Allokieren von temporären Objekten, Mid-life crisis, Memory Leaks) können mit den meisten *Memory Profilern* identifiziert werden.

- Allocation Recording
  Analyse an welcher Stelle im Code (Funktion) Objekte allokiert wurden und welche Beziehungen bzw. Referenzen zwischen den Objekten bestehen.

- Garbage Collection Statistic
  Analyse der Garbage Collection (JVM, CLR). Daraus kann das Verhalten des Garbage Collector genauer betrachtet werden. Wie bereits beschrieben, können diese Informationen hinsichtlich der Performance wichtig sein.

2.3.6 Weitere Techniken zur Leistungsanalyse

Bis zu diesem Punkt wurden wichtige Techniken zu Leistungstests-, -messung und -analyse beschrieben. Es gibt jedoch noch einige weitere Möglichkeiten, die Hinweise auf die Leistung oder auf Probleme einer Anwendung geben. In der Quelle [MEI04] werden folgende Techniken zur Leistungsanalyse beschreiben:

- Netzwerk Monitoring

Ein Leistungsengpass muss nicht immer zwingend innerhalb einer Anwendung entstehen. Im Hinblick auf verteilte Anwendungen kann ein Leistungsengpass durchaus durch ein Netzwerk bzw. durch die Infrastruktur verursacht werden.
Für diese Fälle eignen sich sogenannte *Network Monitoring Tools* mit denen die Netzwerk Performance analysiert werden kann. Netzwork *Monitoring Tools* sind unter anderem in der Lage den Netzwerkverkehr mitzuschneiden und zu analysieren, die Transferrate sowie Zustände verschiedener relevanter Kontenpunkte zu bestimmen.

- Analyse von Log Files

Die Analyse von Log Files kann sehr hilfreich sein, um mögliche Performance Probleme aufzudecken. Interessant für die Analyse sind anwendungsspezifische Log Files sowie die System Log Datei (Windows Ereignisanzeige), in der anwendungsübergreifend (Betriebssystem, IIS, SQL Datebank, Anwendungen etc.) Ereignisse festgehalten werden.

- Instrumentierung

Eine weitere Möglichkeit besteht in der manuellen Instrumentierung der Anwendung. Die Instrumentierung kann in Form von Konsolen-Ausgaben (Logger), einem Event Tracing (eigenes Log File, Windows Ereignisanzeige) oder auch eigenen Leistungsmetriken (PerformanceCounter) erfolgen. Der Vorteil ist sicherlich die Flexibilität mit der Informationen zum Leistungsverhalten gesammelt werden können.

2.3.7 Optimierung

Grundsätzlich gibt es verschiedene Möglichkeiten Optimierungen an einem System bzw. einer Anwendung vorzunehmen. In Abhängig von den Testergebnissen und der Analyse können Optimierungen am Netzwerk, am System, an der Plattform und in der Anwendung vorgenommen werden. Nach der Quelle [MEI04] gibt es dabei folgende Ansätze zur Optimierung.

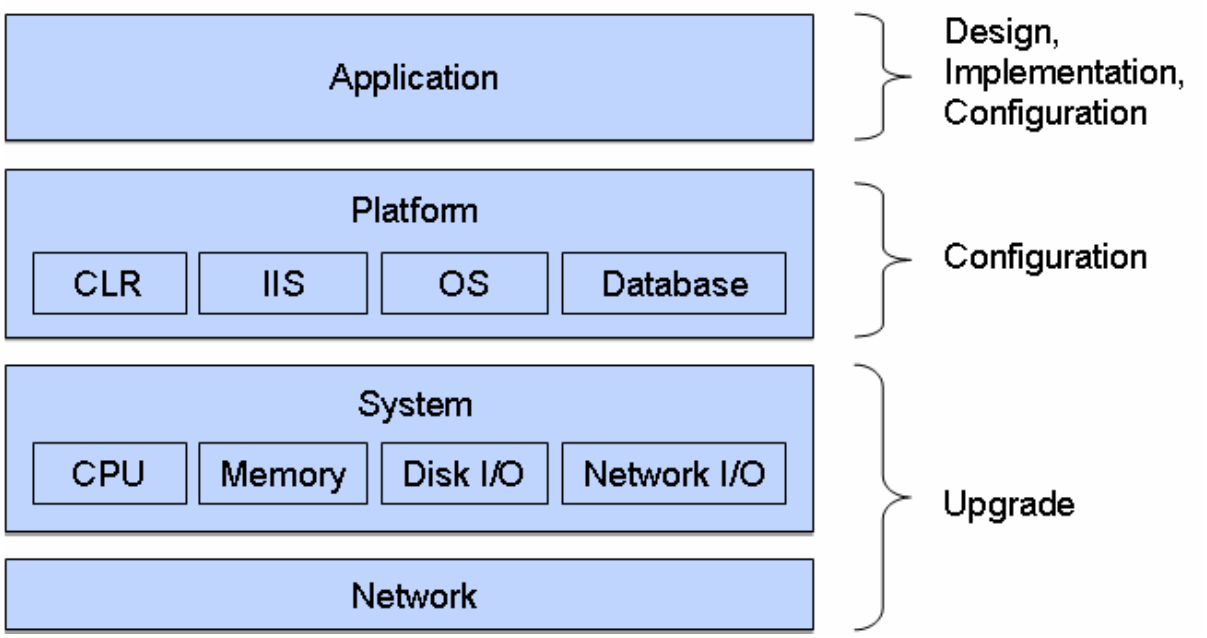


Abb. 2.3.7.1: Möglichkeiten zur Optimierung eines Systems.

- System

Generell kann bei Auffinden eines Leistungsengpasses eine Optimierung bzw. eine Aufrüstung des Systems vorgenommen werden. Das System umfasst Hardwareressourcen wie CPU, Arbeitsspeicher, physikalischer Datenträger sowie Netzwerkschnittstelle.

Wenn sich beispielsweise herausstellt, dass die System-Ressourcen wie z.B. CPU und Arbeitsspeicher zu knapp bemessen sind und die Anwendung selbst keinen Leistungsengpass verursacht, dann macht es durchaus Sinn, das System bzw. die Hardware aufzurüsten. Entsteht der Leistungsengpass aber durch ein ineffizientes Design oder eine ineffiziente Implementierung der Anwendung, dann macht die Optimierung durch Aufrüstung des Systems

keinen Sinn. Denn im Prinzip besteht das zugrundeliegende Problem des Leistungsengpasses in der Anwendung weiterhin, nur wird das Problem gewissermaßen vertuscht.

- Platform

Weiterhin besteht die Möglichkeit Optimierungen an der Plattform vorzunehmen. Die Plattform umfasst Komponenten wie Laufzeitumgebung (CLR, JVM), Web-Server (IIS, Apache), Betriebssystem (Windows, Linux) und Datenbanken (SQL). Optimierungen können hier in Form von Konfigurationsänderungen oder auch dem Austausch einer Komponente erfolgen.

Stellt sich beispielsweise die verwendete Datenbank als Leistungsengpass heraus, kann diese durch eine leistungsfähigere Datenbank ausgetauscht werden. Zudem bieten Plattform Komponenten wie Laufzeitumgebung, Web-Server, Betriebssystem zahlreiche Konfigurationsmöglichkeiten, mit der die Leistungsfähigkeit gesteigert werden kann.

- Application

Optimierungen können auch in der Anwendung selbst vorgenommen werden. Hier entstehen Leistungsengpässe meist durch ein schlechtes Design oder einer schlechten Implementierung. Optimierungen können in Form von Konfigurationsänderungen (Parametrierung), durch Korrektur des Designs oder der Implementierung vorgenommen werden.

Ist ein Leistungsengpass auf die Anwendung zurückzuführen, sollten auch an dieser Ebene Optimierungen vorgenommen werden. Allerdings ist das nicht immer einfach. Änderungen an der Konfiguration der Anwendung sind sicherlich ohne weiteres möglich, nachträgliche Änderungen an der Implementierung bzw. Design sind mit einem entsprechenden Aufwand verbunden.

- Netzwerk

Prinzipiell kann ein Leistungsengpass auch durch das dazwischen liegenden Netzwerk verursacht werden. Das Netzwerk ist natürlich nur für Anwendungen relevant, die auch über ein Netzwerk kommunizieren. Leistungsengpässe im Netzwerk können beispielsweise durch eine zu geringe Bandbreite oder eine zu hohe Auslastung (konkurrierende Zugriffe) entstehen.

3 Entwicklung der Anwendung zur Leistungsanalyse

Die Entwicklung der Anwendung zur Leistungsanalyse umfasst eine Simulation der Benutzerlast, eine Messung von Leistungsmetriken des Flugdatenservers sowie eine Dummy-Komponente des Flugdatenservers (Hüllentest). Nachfolgend wird die Entwicklung dieser Anwendung zur Leistungsanalyse beschrieben.

3.1 Anforderungen

Im Vorfeld der Entwicklung wurde ein Lastenheft erstellt, in dem sämtliche Anforderungen festgehalten wurden. Die wichtigsten Anforderdung daraus werden nun beschrieben.

- Anforderung 1 (Req 2.1-110, Req 3.10)

  Die Anwendung zur Leistungsanalyse soll wahlweise das Gesamtsystem (*ATC.Server* und *ATC.Webserver*) oder nur ein Teil des Systems (*ATC.Server)* analysieren. Hierfür soll die Anwendung in zwei verschiedenen Modi (*Gesamtsystem* und *Teilsystem*) ausgeführt werden können. Des Weiteren soll bei der Analyse des *Gesamtsystems* einen Hüllentest durchgeführt werden, bei dem der *ATC.Server* durch einen *Dummy* ersetzt wird. In der nachfolgenden Abbildung 3.1.1 werden die verschiedenen Modi dargestellt.

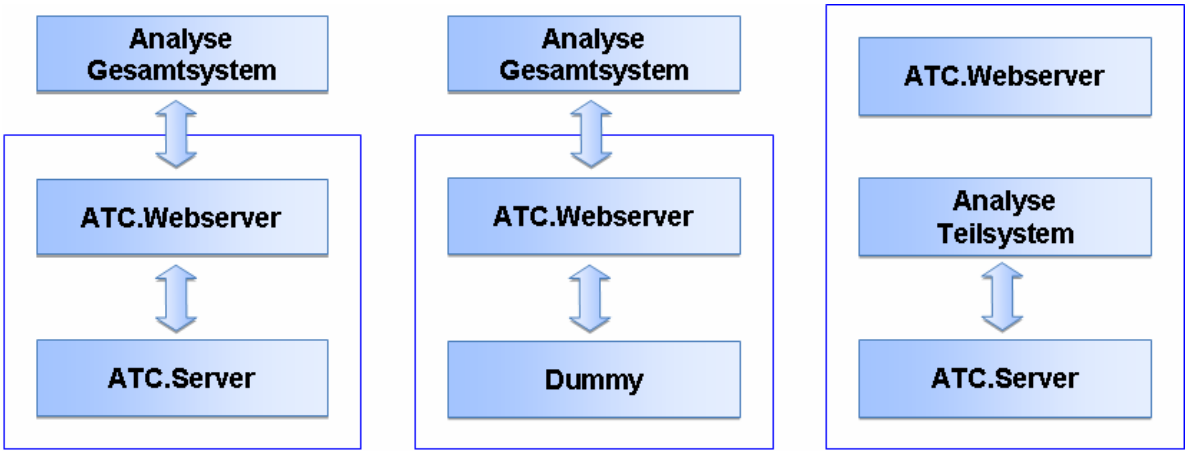


Abb. 3.1.1: Darstellung der verschiedenen Modi der Leistungsanalyse.

Mit der Durchführung der Leistungsanalyse in den verschiedenen Modi (*Gesamtsystem* oder *Teilsystem*), kann ein möglicher Leistungsengpass direkt einer der beiden Flugdatenserver-Komponenten (*ATC.Server* oder *ATC.Webserver*) zugeordnet werden. Bei der Analyse des Gesamtsystems wird für die *Simulation der Benutzerlast* der Web-Service der *ATC.Webserver-Komponente* genutzt und bei der Analyse des Teilsystems wird der *.NET Remoting TcpChannel* der *ATC.Server-Komponente* genutzt. Diese beiden Analyse-Modi sind möglich, da der Flugdatenserver aus den in Abschnitt 2.2.2 beschriebenen getrennten Prozessen besteht und beide über separate Kommunikationsschnittstellen verfügen. Mit dem Hüllentest kann zudem das Verhalten der *ATC.Webserver-Komponente* noch exakter bestimmt werden kann.

- Anforderung 2 (Req 2.1-70)

  Zur Leistungsanalyse sollen während einer *Simulation der Benutzerlast*, prozess-bezogen (*ATC.Server* und *ATC.Webserver*) aktuelle Leistungsmetriken des Flugdatenservers (u.a. CPU- und Arbeitsspeicher-Auslastung) ermittelt werden. Die Intervalllänge bzw. Granularität der Messwertaufnahme soll im Bereich von 25 ms bis 5000 ms konfigurierbar sein.

Mit der Ermittlung dieser Leistungsmetriken kann das Verhalten des Flugdatenserver bewertet und mögliche Leistungsengpässe identifiziert werden.

- Anforderung 3 (Req 2.1-90)

  Während der Durchführung einer Leistungsanalyse sollen die Antwortzeiten des Flugdatenserver (Responses) auf die Client-Anfragen (Requests) der simulierten Benutzerlast ermittelt und kategorisiert gespeichert werden. Die Kategorisierung der Antwortzeiten soll in der folgenden Form erfolgen: alle Antwortzeiten, werden nach einem zeitlichen Intervall kategorisiert, z.B. „100 Antwortzeiten liegen im Bereich von 1 ms bis 50 ms“.

Die Antwortzeiten der Client-Anfragen stellen eine wichtige Leistungsmetrik dar, um einen Leistungsengpass bei einer bestimmten Benutzerlast festzustellen, da wie in Abschnitt 2.3.4 beschrieben, ein enger Zusammenhang zwischen der Benutzerlast und den Antwortzeiten des Flugdatenservers besteht.

- Anforderung 4 ([Req 2.1-10], [Req 2.1-20], [Req 2.1-30])

  Eine Leistungsanalyse *soll* konfigurierbar sein in der Laufzeit, der Anzahl der zu simulierenden Benutzer (bis zu 1000 Benutzer), der Update-Rate (Aktualisierungsintervall) in dem die Client-Anfragen erfolgen und in der Delay-Einstellung (mit oder ohne Delay).

Die Laufzeit der Leistungsanalyse entspricht der Dauer eines Analysedurchlaufes. In dieser Zeit werden die beiden Vorgänge (*Simulation der Benutzerlast* und *Leistungsmessung*) durchgeführt. Mit der Anzahl der zu simulierenden Benutzer wird festgelegt, wie viele Benutzer gleichzeitig simuliert werden. Die Update-Rate entspricht dem Intervall mit welchem ein einzelner simulierter Benutzer Anfragen an den Flugdatenserver stellt. Die Delay-Einstellung entscheidet darüber, mit welcher Berechtigung der simulierte Benutzer ausgeführt wird (als *Standard-Web-Client* oder als *Privilegierter-Web-Client*). Wie bereits in Abschnitt 2.2.4 beschrieben, besteht in den beiden Berechtigungen ein großer Unterschied in der Bereitstellung der Flugzeugnachrichten durch den Flugdatenserver.

Die Durchführung einer Leistungsanalyse mit einer bestimmten Konfiguration entspricht einem Analyselauf. Grundsätzlich soll es möglich sein, beliebig viele solcher Analyseläufe nacheinander und unabhängig voneinander ausführen zu können. Die Konfiguration (Parameter) eines Analyselaufes, kann manuell eingegeben oder automatisch aus einer XML-Datei eingelesen werden können. In dieser XML-Datei können beliebig viele Konfigurationen für Analyseläufe (Testskript) hinterlegt werden.

3.2 Entwurfsalternativen

Für die Realisierung der Anwendung bieten sich verschiedene Entwurfsalternativen an, die nachfolgend beschrieben und bewertet werden.

3.2.1 Alternativen zur Simulation der Benutzerlast

Die Anwendung zur *Simulation der Benutzerlast* muss in der Lage sein, eine beliebige Anzahl an Benutzern bzw. Client-Anfragen simulieren zu können. Für die Leistungsanalyse des Flugdatenservers sind bis zu 1000 zu simulierende Benutzer vorgesehen, es können aber auch mehr Benutzer simuliert werden. Weiterhin muss die Anwendung die Antwortzeiten jeder durchgeführten Client-Anfrage messen. Für die Realisierung der Anwendung zur *Simulation der Benutzerlast* haben sich die in Abbildung 3.2.1.1 dargestellten zwei Alternativen angeboten.

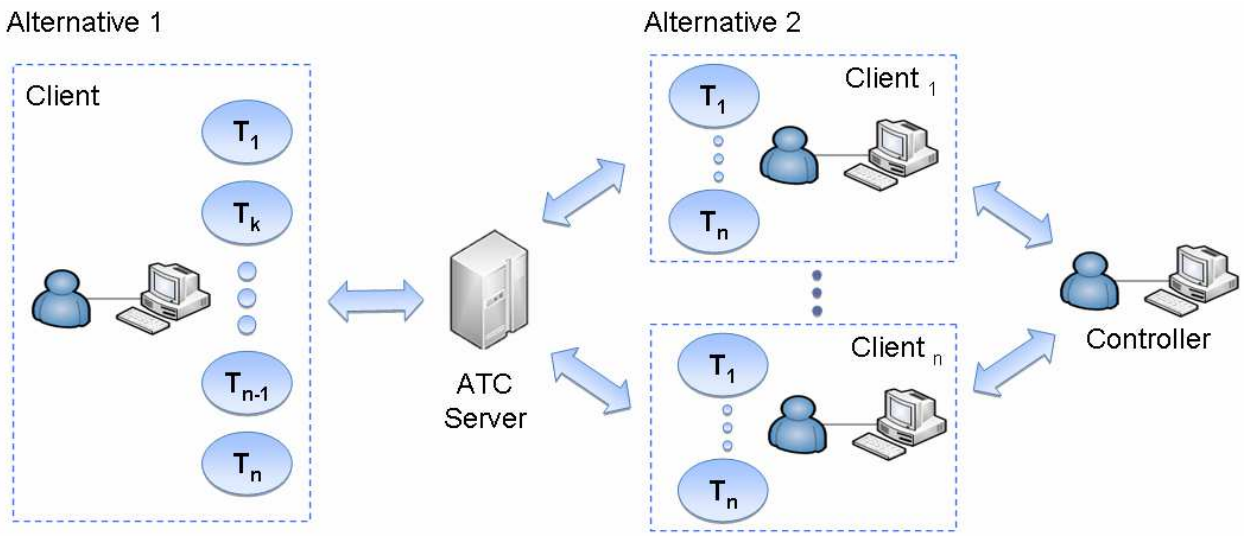


Abb. 3.2.1.1: Entwurfsalternativen zur Simulation der Benutzerlast.

Alternative 1

In *Alternative 1* realisiert eine Arbeitsstation (Client) bzw. ein Prozess die *Simulation der Benutzerlast* über Multi-Threading. Jeder zu simulierende Benutzer wird demnach über einen separaten Thread simuliert, der kontinuierlich Client-Anfragen an die entsprechende Kommunikationsschnittstelle, abhängig des Analyse-Modus (*Gesamtsystem* oder *Teilsystem*), stellt. Die Messung der Antwortzeiten auf die Client-Anfragen erfolgt innerhalb des jeweiligen Threads.

Vorteile:

- Für die *Simulation der Benutzerlast* wird nur eine Arbeitsstation (Client) benötigt.
- Geringe Komplexität in der Implementierung.

Nachteil:

- Bei einer bestimmten Menge bzw. Schwelle an zu simulierenden Benutzern (Thread-Anzahl) wird die Messung der Antwortzeiten auf die Client-Anfragen verfälscht. Beim Testen dieser Entwurfsalternative wurden die gemessenen Antwortzeiten bereits bei einer Anzahl von 200 zu simulierenden Benutzern (Threads) verfälscht. Diese Verfälschung ergibt sich aus dem typischen Verhalten einer großen Menge konkurrierender Threads, bei der in jedem Fall verlängerte Wartezeiten entstehen. Dadurch wird die Messung der Antwortzeiten der Client-Anfragen um die jeweilige Wartezeit verfälscht.

Alternative 2

In Alternative 2 dagegen, wird eine beliebige Menge an Arbeitsstationen (Clients) eingesetzt, auf die die Gesamtanzahl der zu simulierenden Benutzer verteilt wird. Auf jeder einzelnen Arbeitsstation wird dabei die *Simulation der Benutzerlast* ausgeführt. Die der Arbeitsstation zugewiesene jeweilige Benutzeranzahl wird dabei wie in *Alternative 1* auch über Multi-Threading realisiert. Die Messung der Antwortzeiten erfolgt hier ebenfalls innerhalb des jeweiligen Threads.

Durch den Einsatz mehrerer Arbeitsstationen ist jedoch eine zusätzliche Komponente (Controller) nötig, um die *Simulation der Benutzerlast* bzw. die Arbeitsstationen zu koordinieren. Die Koordination umfasst dabei unter anderem das Senden eines gemeinsamen Startzeitpunktes für die Simulation (Start-Synchronisation) sowie das Einsammeln der gemessen Antwortzeiten auf der jeweiligen Arbeitsstation am Ende einer Simulation.

Alternative 2 ist eine komplexe verteilte Anwendung, in der ISC-Mechanismen (Inter-Server Communication) benötigt werden.

Vorteil:

- Die zu simulierende Benutzerlast wird auf mehrere Arbeitsstationen verteilt. Dadurch wird vermieden, dass auf einer einzelnen Arbeitsstation die in *Alternative 1* genannte Schwelle der Benutzerlast überstiegen und damit die Messung der Antwortzeiten verfälscht wird.

Nachteil:

- Höhere Komplexität in der Implementierung
  Da es sich um eine verteilte Anwendung mit einer Inter-Server-Kommunikation handelt, ist die Implementierung deutlich komplexer und damit aufwändiger.

Entscheidung

Eine korrekte Ermittlung der Antwortzeiten ist für die Identifizierung von Leistungsengpässen sehr wichtig. Für die Realisierung der *Simulation der Benutzerlast* hat sich *Alternative 2* (verteilte Anwendung) deshalb als die bessere herausgestellt. Ausschlaggebend hierfür ist der Vorteil der unverfälschten Messwertaufnahme der Antwortzeiten der Client-Anfragen. Deshalb überwiegt dieser Vorteil ganz klar gegenüber dem Nachteil der höheren Komplexität in der Implementierung.

3.2.2 Alternativen zur Messung der Leistungsmetriken

Wie in den Anforderungen beschrieben, müssen während einer Leistungsanalyse zusätzlich zur *Simulation der Benutzerlast* verschiedene Leistungsmetriken wie CPU-, Arbeitsspeicher-, Netzwerkschnittstellen-Auslastung gemessen werden. Für die Messung dieser Leistungsmetriken bieten sich ebenfalls zwei Alternativen an.

Alternative 1

Bei der ersten Alternative werden die benötigten Leistungsmetriken über die Windows Anwendung *PerformanceMonitor* (perfmon.exe) während einer Leistungsanalyse gemessen. Mit diesem Tool können sämtliche auf dem System zur Verfügung stehenden Leistungsmetriken über einen definierten Zeitraum gemessen werden. Die Messung mit dem *PerformanceMonitor* muss allerdings, mit jedem Start einer Leistungsanalyse, manuell neu angestoßen werden.

Vorteil:

- Kein Aufwand in Form von zusätzlicher Implementierung einer eigenen Anwendung zur Messung der Leistungsmetriken.

Nachteile:

- Die gesammelten Messwerte liegen dezentral in einer externen Anwendung (*PerformanceMonitor*) vor. Die gemessenen Leistungsmetriken werden in Form von Diagrammen und Berichten dokumentiert. Zwar können die Messwerte gespeichert oder gedruckt werden, aber eine Zuordnung zu den Parametern, unter denen ein Analyselauf durchgeführt wird, ist nur mit einem hohen manuellen Aufwand möglich. So geht der Zusammenhang zwischen den gemessenen Leistungsmetriken und den dazugehörigen Parametern eines Analyselaufes (Anzahl zu simulierende Benutzer, Update-Rate, Delay-Einstellung, Laufzeit etc.) verloren. Weiterhin können die Messwerte in keiner Weise für eine Analyse z.B. in Microsoft Excel verwendet werden.

- Die Messung der Leistungsmetriken im *PerformanceMonitor* kann maximal nur über einen Zeitraum von 1000 Sekunden (ca. 16,7 min) erfolgen. Für die Durchführung von *Load-* oder *Stress-Tests* werden jedoch wesentlich längere Zeiträume (Stunden oder Tage) zur Leistungsmessung benötigt.

- Hoher manueller Aufwand bei der Zusammenführung der Messdaten und der Konfiguration eines Analyselaufes.

Alternative 2

Als zweite Alternative würde sich zur Leistungsmessung eine eigene, entwickelte Anwendung anbieten. Wie bereits in Abschnitt 2.3.3 beschrieben, ist es möglich die verschiedenen Leistungsmetriken über das *.NET-Framework* abzurufen. Das *.NET-Framework* bietet die Möglichkeit die Leistungsmetriken über die Klasse *PerformanceCounter* (Namespace *System.Diagnostics)* zu verwenden. Damit können ebenfalls wie in *Alternative 1* sämtliche Leistungsmetriken die auf dem System zur Verfügung stehen, während einer Analyse gemessen werden. Der gesamte Ablauf der Messung der Leistungsmetriken kann durch eine Integration in die verteilte Anwendung automatisiert werden.

Vorteile:

- Hohe Flexibilität, wie die Leistungsmessung realisiert wird.
- Die eigene Leistungsmessung kann damit als Komponente in die verteilte Anwendung zur Leistungsanalyse integriert werden. Somit kann der Ablauf der Leistungsmessung ohne manuellen Aufwand automatisiert durchgeführt werden. Zusätzlich liegen die Messergebnisse zentral in der eigenen Anwendung vor.

Nachteile:

- Verteilte Anwendung (*Simulation der Benutzerlast* und *Leistungsmessung*) wird komplexer.
- Zusätzlicher Implementierungsaufwand

Entscheidung

Zur Realisierung der Leistungsmessung hat sich die zweite Alternative als die bessere herausgestellt. Es entsteht zwar ein zusätzlicher Implementierungsaufwand, allerdings wird die Leistungsmessung hinsichtlich eigener Anforderungen viel flexibler. Durch die Integration in die verteilte Anwendung zur Leistungsanalyse und dem damit möglichen automatischen Ablauf der Leistungsmessung, besteht kein zusätzlicher manueller Aufwand bei der Leistungsmessung und der Datenbereitstellung.

3.3 Grobentwurf

Im Grobentwurf werden der Aufbau, der Ablauf und die Kommunikation innerhalb der verteilten Anwendung beschrieben. Nachfolgend wird der Aufbau bzw. die verschiedenen Komponenten der Anwendung beschrieben.

3.3.1 Aufbau der Anwendung

Die ausgewählten Entwurfsalternativen führen zu einer Struktur einer verteilten Anwendung. Eine verteilte Anwendung zeichnet sich dadurch aus, dass mehrere Prozesse auf verschiedenen Systemen verteilt sind und an der Lösung einer gemeinsamen Aufgabe arbeiten. Diese Definition trifft auf die Anwendung zur Leistungsanalyse zu, weil diese aus mehreren Komponenten besteht und verteilt an der Leistungsanalyse arbeitet.

Die verteilte Anwendung besteht aus den Komponenten *Controller*, *Worker*, *Performance* und *ServerDummy* (Hüllentest). Jede Komponente wird in einem separaten Prozess (Executable) realisiert. Nachfolgend werden die einzelnen Komponenten genauer beschrieben.

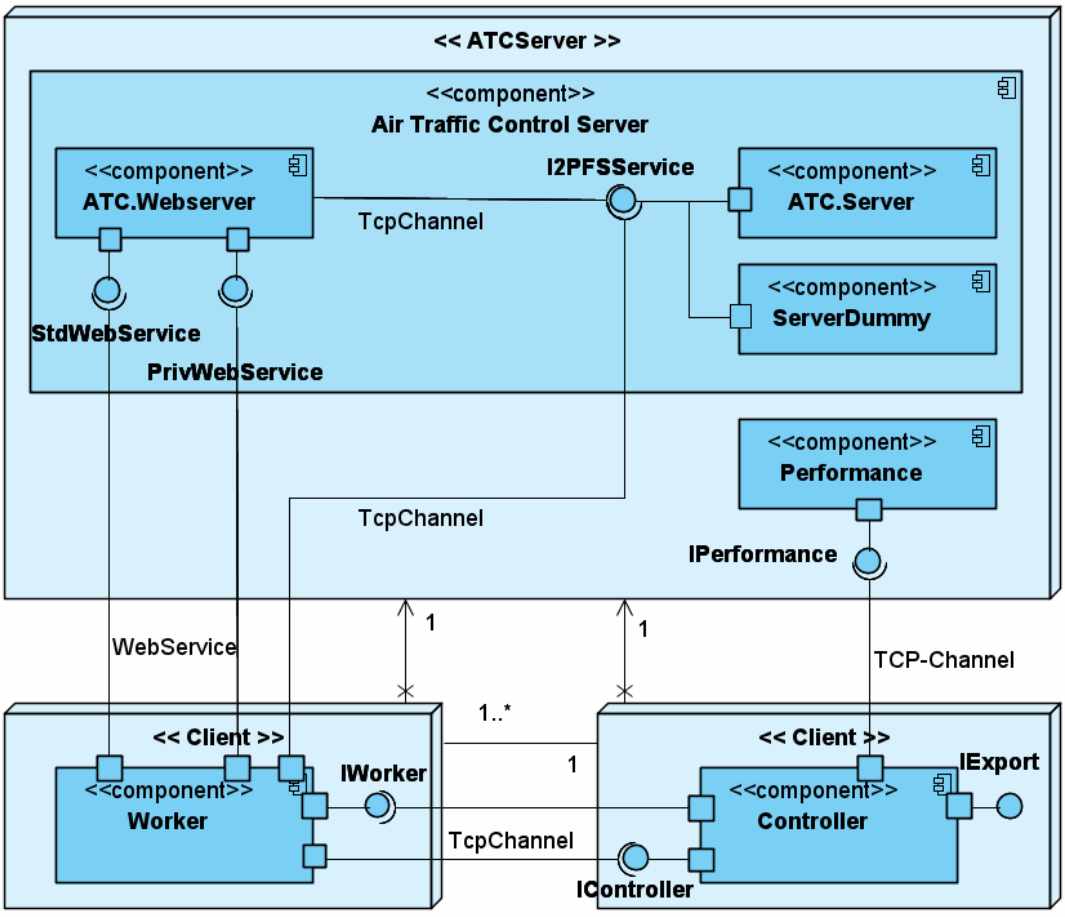


Abb. 3.3.1.1: UML-Verteilungsdiagramm der verteilten Anwendung.

Komponente Controller

Der *Controller* koordiniert die gesamte Leistungsanalyse. Die Koordination umfasst die Steuerung der Komponenten *Worker* und *Performance* und teilt sich in die folgenden Aufgaben auf:

- Das Einlesen der Konfiguration für einen Analyselauf.
- Das Starten der *Simulation der Benutzerlast* auf der *Worker-Komponente*.
- Das Starten der *Leistungsmessung* auf der Performance-Komponente.
- Das Einsammeln der Messwerte nach Beendigung eines Analyselaufes.

Das Einlesen der Konfiguration einer *Leistungsanalyse* umfasst die in Abschnitt 3.1. beschriebenen Parameter für einen Analyselauf. Zudem wird ein zusätzlicher Parameter (Ja/Nein) für die Ausführung eines Logging-Mechanismus eingelesen. Der Logging-Mechanismus protokolliert wichtige Aktionen innerhalb der verteilten Anwendung in einer TXT-Datei. Diese kann zu Analysezwecken oder im Fehlerfall ausgewertet werden.

Der *Controller* wird zur Steuerung der Leistungsanalyse auf einem beliebigen Client im Netzwerk ausgeführt. Um die *Simulation der Benutzerlast* zu steuern, kommuniziert der *Controller* mit beliebig vielen *Worker-Komponenten,* wie in Abbildung 3.3.1.1 dargestellt. Die Kommunikationsparameter der *Worker* werden über eine XML-Datei konfiguriert. Die *Worker* können dabei auf beliebig vielen Clients im Netzwerk ausgeführt werden. Für die Funktionalität der bidirektionalen Kommunikation mit dem *Worker,* implementiert der *Controller* das Interface *IController* (*Worker* zu *Controller*) und nutzt das Interface *IWorker* (*Controller* zu *Worker*).

Zudem kommuniziert der *Controller* mit genau einer *Performance-Komponente* die sich auf dem Flugdatenserver befindet, wie in Abbildung 3.3.1.1 dargestellt. Die Kommunikationsparameter der *Performance-Komponente* sind fest in den *Controller* kodiert. Für die Kommunikation mit der *Performance-Komponente* wird das Interface *IPerformance* verwendet.

Komponente Worker

Der *Worker* realisiert die *Simulation der Benutzerlast* und die Messung der Antwortzeiten auf die Client-Anfragen. Die Benutzer werden simuliert, indem Client-Anfragen an den Flugdatenserver gestellt werden. Jeder zu simulierende Benutzer führt demnach in einem vorgegeben Intervall (Update-Rate) Anfragen an den Flugdatenserver aus. Wird das *Gesamtsystem* analysiert, werden die Client-Anfragen an den Web-Service des *ATC.Webservers* gestellt. Wird *das Teilsystem* analysiert, werden die Client-Anfragen direkt an den *TcpChannel* des *ATC.Servers* gestellt. Zusätzlich zur *Simulation der Benutzerlast* werden die Antwortzeiten der Client-Anfragen gemessen und gesammelt.

Wie bereits im Abschnitt 3.2.1 beschrieben, sieht die ausgewählte Entwurfsalternative vor, dass mehrere Clients für die *Simulation der Benutzerlast* verwendet werden. Deshalb wird der *Worker* auf beliebig vielen Clients gleichzeitig ausgeführt. Der *Worker* kommuniziert mit dem Flugdatenserver (*Web-Service* und *TcpChanne*l) und dem *Controller* der ihn steuert. Die Kommunikationsparameter für die Kommunikation mit dem Flugdatenserver sind in den *Worker* fest kodiert. Die Kommunikationsparameter für die Kommunikation mit dem *Controller* erhält jeder *Worker* einmalig mit dem Start des *Controllers*. Mit dem Start des *Controllers* macht sich dieser bei allen *Worker-Komponenten* bekannt, damit die bidirektionale Kommunikation gegeben ist. Für die Kommunikation mit dem *Controller* wird das Interface *IController* genutzt. Für die Kommunikation mit dem Flugdatenserver dagegen, wird in Abhängigkeit des Analyse-Modus der *Web-Service* oder das Interface *I2PFSService* des Flugdatenservers genutzt.

Komponente Performance

Die *Performance-Komponente* realisiert die Messung der verschiedenen Leistungsmetriken, wie CPU- und Arbeitsspeicher-Auslastung, während einer Leistungsanalyse. Grundsätzlich kann die *Performance-Komponente* beliebig viele Leistungsmetriken während einer Leistungsanalyse messen. Die zu messenden Leistungsmetriken werden über eine XML-Datei konfiguriert. Da die Leistungsmetriken des Flugdatenserver gemessen werden, wird die *Performance-Komponente* direkt auf dem Flugdatenserver ausgeführt. Für die Kommunikation vom *Controller* zur *Performance-Komponente,* wird das Interface *IPerformance* durch die *Performance-Komponente* implementiert. Hierfür reicht eine unidirektionale Kommunikation aus.

Komponenten ServerDummy

Der *ServerDummy* wird für die Durchführung eines *Hüllentests* zur *Leistungsanalyse* des *Gesamtsystems* benötigt. Hierfür wird der *ATC.Server* durch den *ServerDummy* ersetzt, der ein erwartetes Verhalten bzw. Funktionalität nachbildet. Der *ServerDummy* realisiert nur die Funktion der Rückgabe einer bestimmten konfigurierbaren Menge an Flugzeugnachrichten. Die Sonstige sämtliche Funktionalität der *ATC.Server-Komponente* bleibt dabei unberücksichtigt. Der *ServerDummy* hat somit eine wesentlich reduziertere Funktionalität als der *ATC.Server*. Für die Kommunikation vom *ATC.Webserver* zum *ServerDummy* wird das Interface *I2P2SService* des Flugdatenservers implementiert.

Mit dem *Hüllentest* kann genauer bestimmt werden, inwiefern sich die Kommunikation der beiden Komponenten *ATC.Webserver* und *ATC.Server* bzw. *DummyServer* auf die Leistung des Flugdatenservers auswirkt.

3.3.2 Ablauf der Anwendung

In der Abbildung 3.3.2.1 wird der Ablauf und die Kommunikation der verschiedenen Komponenten untereinander der verteilten Anwendung zur Leistungsanalyse beschrieben.

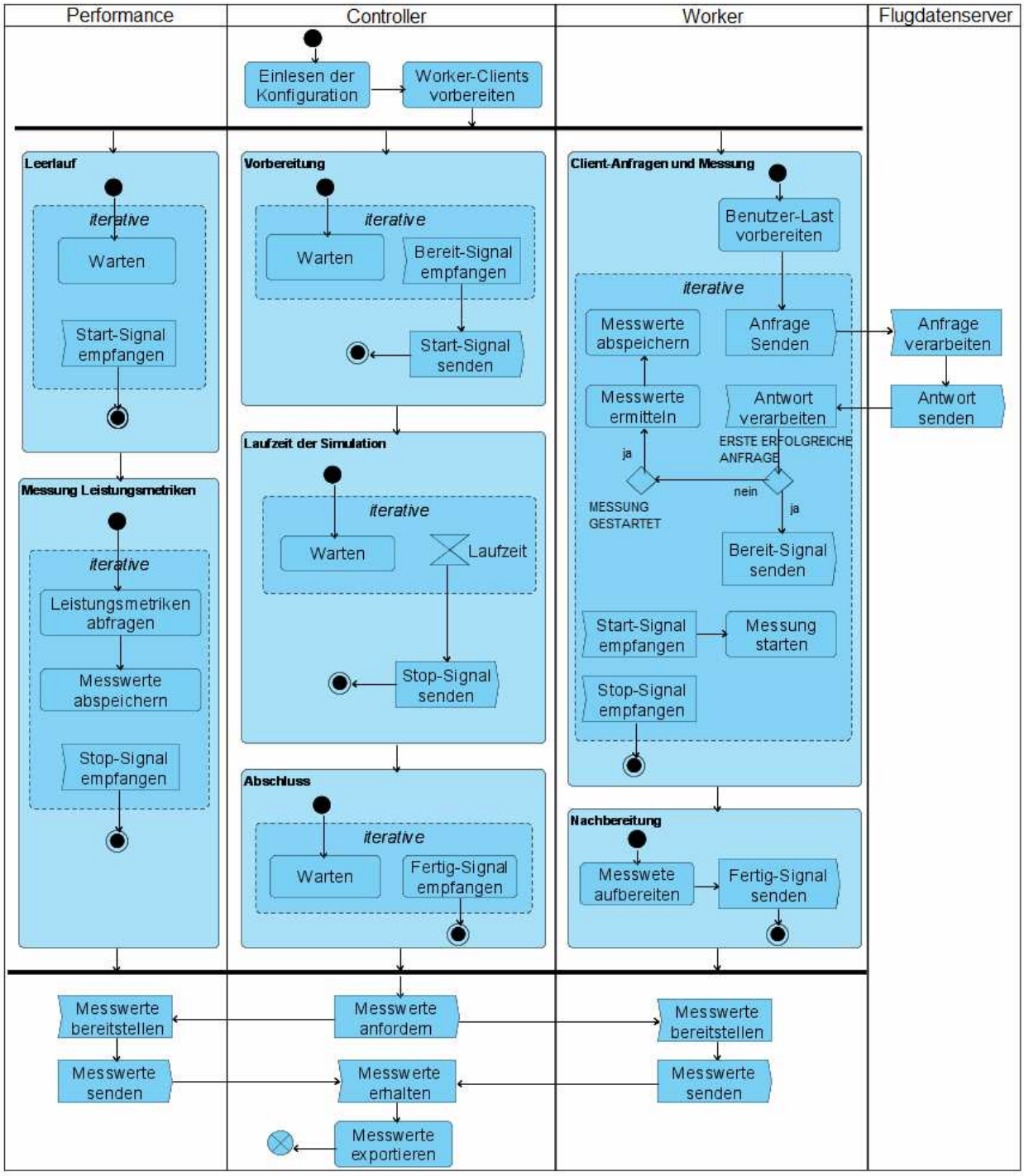


Abb. 3.3.2.1: UML-Aktivitätsdiagramm zum Ablauf der verteilten Anwendung.

Der Ablauf der verteilten Anwendung, wie in Abbildung 3.3.2.1 dargestellt, kann in die nachfolgenden fünf Phasen eingeteilt werden:

- Phase 1: Einlesen der Konfiguration

  Das Einlesen der Konfiguration kann manuell oder automatisch, wie in Abschnitt 3.1 beschrieben, erfolgen. Die Konfiguration umfasst die Anzahl der zu simulierenden Benutzer, die Upate-Rate (Anfrage-Intervall), die Delay-Einstellung, die Laufzeit eines Analyselaufes sowie einen Parameter für das Logging.

- Phase 2: Vorbereitung der *Worker*

  In der zweiten Phase erfolgt in der *Controller-Komponente* die Berechnung der Verteilung der zu simulierenden Benutzer auf die in der Konfigurationsdatei (XML-Datei) definierten *Worker*. In der Regel, erhält jeder *Worker* die gleiche Anzahl an zu simulierenden Benutzer. Nach dieser Aufteilung werden die verfügbaren *Worker* vom *Controller* über einen asynchronen Aufruf (RPC) informiert, um die Vorbereitung des jeweiligen *Worker* einzuleiten. Alle *Worker* führen diese Vorbereitung parallel aus. Die *Worker* instanziieren in dieser Vorbereitungsphase alle benötigten Ressourcen, die für die *Simulation der Benutzerlast* benötigt werden. In dieser Vorbereitungsphase beginnen die *Worker* zudem direkt mit dem Start der Client-Anfragen an den Flugdatenserver. Die Messwerte (z.B. Antwortzeiten) werden jedoch in der gesamten Vorbereitungsphase noch nicht ermittelt. Die eigene Vorbereitungsphase eines *Worker* wird abgeschlossen, sobald der *Worker* seine erste Client-Anfrage an den Flugdatenserver erfolgreich gestellt und beantwortet (Request und Response) bekam. Unmittelbar danach wird der *Controller* über den Abschluss der Vorbereitungsphase vom jeweiligen *Worker* informiert. Der *Controller* bleibt dabei solange im Wartezustand bis alle *Worker* ihre Vorbereitungsphase abgeschlossen haben. Dieser Wartezustand wird durch einen Synchronisationsmechanismus im *Controller* realisiert. Die Vorbereitungsphase auf dem *Worker* dient zur Sicherstellung, dass alle *Worker* in der Lage sind, den Flugdatenserver für die anstehende Leistungsanalyse zu erreichen. Der Synchronisationsmechanismus wiederum sorgt dafür, dass die nächste Phase (Messung) für alle gleichzeitig eingeleitet wird. Ansonsten würden zu starke Unterschiede im Zeitverhalten bei der Messwertaufnahme entstehen.

- Phase 3: Messwertaufnahme

  In der dritten Phase wird die Messwertaufnahme durch die *Performance-Komponente* und die *Worker-Komponenten* gestartet. Hierfür sendet der *Controller* ein Startsignal an *die Performance-Komponente* auf dem Flugdatenserver und an die *Worker-Komponenten*. Diese beginnen mit dem Erhalt des Startsignals mit der Messwertaufnahme. Der *Controller* befindet sich nun solange wieder in einem Wartezustand, bis die zuvor definierte Laufzeit des Analyselaufes vorüber ist. Ist der Analyselauf abgeschlossen, sendet der *Controller* ein Stoppsignal zum Beenden der Messwertaufnahme auf den *Worker-Komponenten* und der *Performance-Komponente*. Auf dem *Worker* wird zusätzlich das Beenden der *Simulation der Benutzerlast* eingeleitet.

- Phase 4: Abschlussphase

  Wie in der zweiten Phase, dient die vierte Phase als Synchronisationsmechanismus für die *Worker*. Der *Controller* befindet sich solange im Wartezustand, bis alle *Worker* die *Simulation der Benutzerlast* und die Aufbereitung der Messwerte abgeschlossen haben. Die Aufbereitung der Messwerte auf einem *Worker* beinhaltet das Zusammenfassen der Messwerte der simulierten Benutzer. Erst nach Abschluss dieser Phase können die zusammengefassten Messwerte der jeweiligen *Worker* durch den Controller eingesammelt werden. Damit wird die letzte Phase eingeleitet.

- Phase 5: Einsammeln und Exportieren der Messwerte

  In der letzten Phase der Leistungsanalyse werden somit die Messwerte, die sich in der *Performance-Komponente* und den *Worker-Komponenten* befinden, durch den *Controller* eingesammelt. Sind alle Messwerte im *Controller* eingegangen, erfolgt eine Zuordnung der Messwerte zu den Konfigurationsparametern, mit der ein Analyselauf ausgeführt wurde. Damit liegen alle Messwerte im *Controller* vor und können exportiert (CSV-Datei) werden.

3.3.3 Kommunikation innerhalb der Anwendung

Wie im vorigen Abschnitt 3.3.2 beschrieben, kommunizieren die Komponenten der verteilten Anwendung untereinander, um eine Leistungsanalyse durchzuführen. In der verteilten Anwendung gibt es somit nachfolgende Kommunikationsbeziehungen:

- Controller zu Worker (Steuerung)
- Worker zu Controller (Synchronisation)
- Controller zu Performance (Steuerung)

Die Kommunikation zwischen den Komponenten wird ebenfalls über den in Kapitel 2.2.3 erwähnten .NET RPC-Mechanismus *TcpChannel* realisiert. Dieser RPC-Mechanismus wird bereits für die Kommunikation zwischen den Flugdatenserver-Komponenten (*ATC.Webserver* und *ATC.Server*) verwendet. Nachfolgend wird die Realisierung der Kommunikation über den *TcpChannel* für die oben genannten Kommunikationsbeziehungen innerhalb der verteilten Anwendung exemplarisch anhand der Kommunikation zwischen dem *Controller* und der *Performance-Komponente* beschrieben.

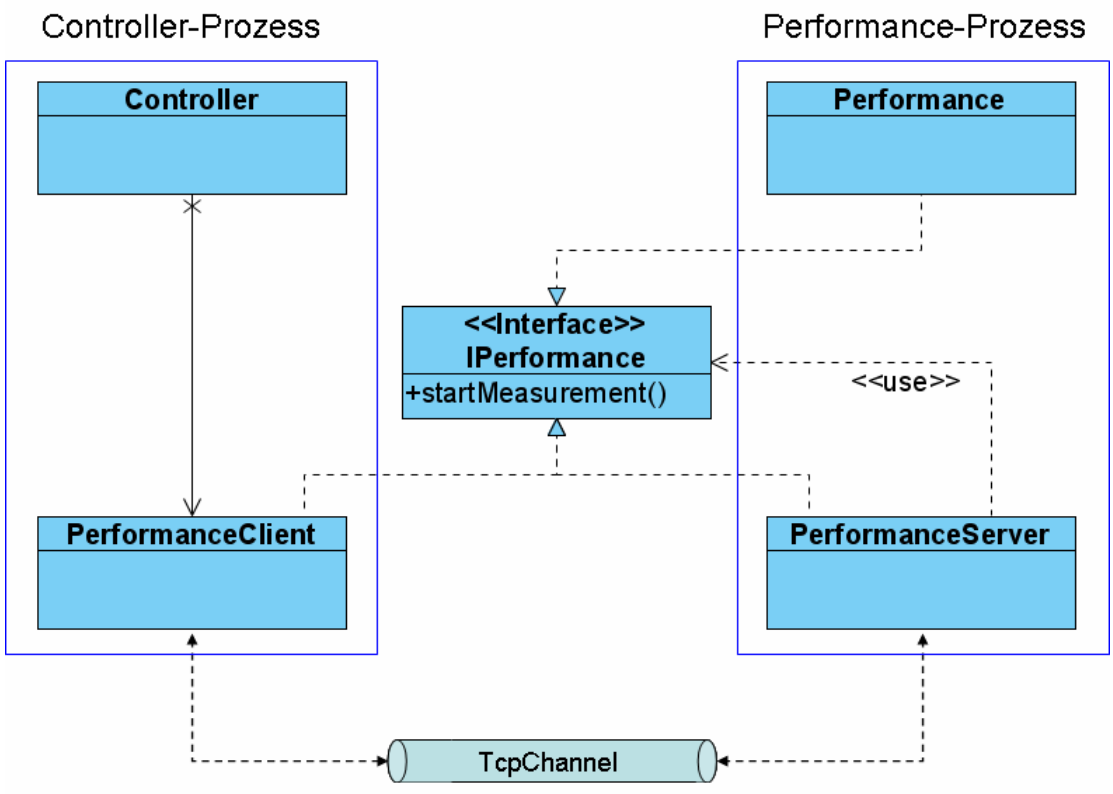


Abb. 3.3.3.1: Schema zur Umsetzung der Kommunikation innerhalb der verteilten Anwendung.

In diesem Beispiel möchte die Klasse *Controller* im *Controller-Prozess* die Methode *startMeasurement()* aufrufen, die sich in der Klasse *Performance* im *Performance-Prozess* befindet. Für diese Kommunikationsbeziehung gibt es in den beiden Prozessen jeweils eine Hilfsklasse (*Stub*). Im *Controller-Prozess* ist dies die Klasse *PerformanceClient* (*ClientStub*). Im *PerformanceProzess* dagegen die Klasse *PerformanceServer* (*ServerStub*). Die Namen dieser Stubs sind so gewählt, dass daraus die Richtung und die Rolle für die Kommunikation ersichtlich ist. In diesen beiden Klassen (*Stubs*) wird jeweils die Funktionalität der Kommunikation mit dem *TcpChannel* ausgelagert. Der *PerformanceClient* realisiert dabei nur die ausgehende Kommunikation zum *Performance-Prozess*, der *PerformanceServer* wiederum die eingehende Kommunikation. Beide Hilfsklassen (*Stubs*) implementieren das Interface *IPerformance*, damit sichergestellt wird, dass in beiden Hilfsklassen dieselben Methoden auch vorhanden sind.

Der *Controller* nutzt den *PerformanceClient,* um auf dem *Performance-Prozess* RPC-Aufrufe durchführen zu können. Der *PerformanceServer* ist ein Proxy-Objekt der die eingehenden Aufrufe an die konkrete Klasse Performance delegiert. Hierfür implementiert und nutzt der *PerformanceServer* das Interface *IPerformance*. In der Klasse *Performance* ist die Methode *getMeasurement()* dann tatsächlich auch ausimplementiert. Die Klasse *Performance* implementiert das Interface *IPerformance*, damit sichergestellt ist, dass der *PerformanceServer* (Proxy-Objekt) delegieren kann.

Mit diesem Ansatz wird nur eine unidirektionale Kommunikation ermöglicht. Für eine bidirektionale Kommunikation wird auf jeder Seite jeweils ein weiterer *Stub* benötigt. Sämtliche der in diesem Abschnitt genannten Kommunikationsbeziehungen werden nach diesem Schema umgesetzt.

3.4 Feinentwurf

Nachfolgend werden die genannten Komponenten aus dem Grobentwurf (Abschnitt 3.3.1) genauer beschrieben. Dabei werden die wichtigsten Bestandteile bzw. Klassen erläutert. Aus Platzgründen handelt es sich bei den dargestellten UML-Diagrammen teilweise um Auszüge aus den vollständigen UML-Diagrammen für die Komponenten. Die vollständigen UML-Diagramme können jedoch dem Anhang entnommen werden.

3.4.1 Komponentenübergreifende Schnittstellen

Nachfolgend werden verschiedene Schnittstellen (Interfaces) beschrieben, die komponentenübergreifend für die Kommunikation zwischen den Komponenten verwendet werden. Damit die Komponenten über ihre *Stubs* kommunizieren können, implementieren diese *Stubs* eine gemeinsame Schnittstelle. Die Schnittstellen realisieren eine gemeinsame Schnittstellendefinition für die jeweiligen *Client*- bzw. *Server-Stubs* in die Komponenten. Für eine bidirektionale Kommunikation zwischen zwei Komponenten, werden daher entsprechend zwei Schnittstellendefinitionen benötigt.

Interface IWorker

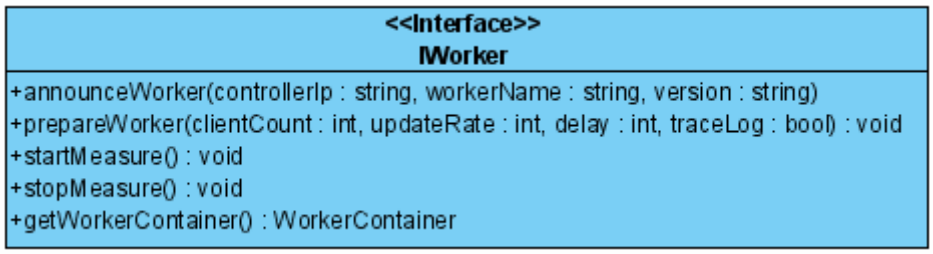


Abbildung 3.4.1.1: UML-Klassendiagramm für das Interface *IWorker*.

Das Interface *IWorker* beinhaltet alle Methoden, die für die Kommunikation zwischen dem *Controller* und dem *Worker* benötigt werden. Das Interface fordert die Implementierung von fünf Methoden (s. Abb. 3.4.1.1). Im *Controller* implementiert der *Client-Stub* das Interface *IWorker*. Im *Worker* dagegen, implementiert und nutzt der *Server-Stub* das Interface *IWorker*.

Mit der Methode *announceWorker()* macht sich der *Controller* einem *Worker* bekannt. Hierfür erfolgt unter anderem die Übergabe seiner IP-Adresse an einen *Worker*. Diese IP-Adresse benötigt ein *Worker* für die Kommunikation in die Rückrichtung zum *Controller*. Des Weiteren

wird jedem *Worker* durch den *Controller* ein eindeutiger Name zur Identifikation zugewiesen. Der dritte übergebene Parameter ist die Versionsnummer der Anwendung zur Leistungsanalyse. Anhand dieser Versionsnummer überprüft ein *Worker* selbst, ob seine Versionsnummer die des *Controllers* entspricht. Sind die beiden Versionsnummern unterschiedlich, wird die Leistungsanalyse mit einer Fehlermeldung abgebrochen. Damit wird verhindert, dass die Leistungsanalyse mit unterschiedlichen Programmversionen ausgeführt wird, was potenziell in einer verteilten Anwendung zu Fehlern führen könnte.

Mit der Methode *prepareWorker()* wird die Vorbereitung der *Simulation der Benutzerlast* durch den *Controller* auf einem *Worker* eingeleitet. Hierbei erfolgt die Übergabe der Konfiguration für eine Simulation. Anhand dieser Konfiguration bereitet jeder *Worker* die Simulation vor.

Mit den Methoden *startMeasure()* und *stopMeasure()* wird die Messwertaufnahme während einer Simulation auf dem *Worker* gestartet bzw. gestoppt. Mit der Methode *getWorkerContainer(),* fordert der *Controller* die gemessen Leistungsmetriken (Antwortzeiten, Anzahl Anfragen etc.) nach dem Ende einer Simulation von einem *Worker* an.

Interface IController

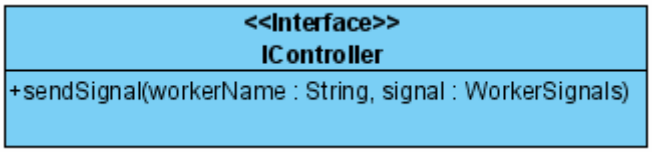


Abbildung 3.4.1.2: UML-Klassendiagramm für das Interface *IController*.

Das Interface *IController* beinhaltet eine Methode, die für die Kommunikation zwischen einem *Worker* und dem *Controller* benötigt wird. Dieses Interface fordert die Implementierung der Methode *sendSignal()*. Im *Worker* implementiert der *Client-Stub* das Interface *IController*. Im *Controller* dagegen, implementiert und nutzt der *Server-Stub* das Interface *IController*.

Mit der Methode *sendSignal()* sendet ein *Worker* Synchronisationssignale an den *Controller* zurück. Hat ein *Worker* die Vorbereitung einer *Simulation der Benutzerlast* abgeschlossen, dann sendet er das Signal *READY*. Nach der Beendigung einer Simulation wiederum, versendet der *Worker* das Signal *FINISH*. Diese Signale werden für den in Abschnitt 3.3.2

erwähnten Synchronisationsmechanismus benötigt. In Abschnitt 3.4.3 wird auf diesen Mechanismus noch genauer eingegangen.

Interface IPerformance

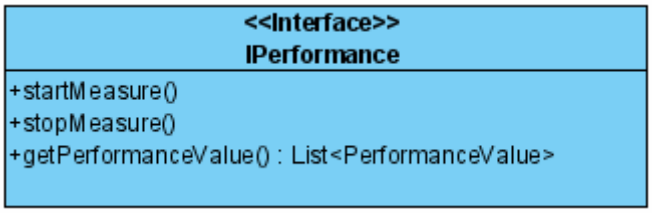


Abbildung 3.4.1.3: UML-Klassendiagramm für das Interface *IPerformance.*

Das Interface *IPerformance* beinhaltet alle Methoden, die für die Kommunikation zwischen dem *Controller* und der *Performance-Komponente* benötigt werden. Das Interface fordert die Implementierung von drei Methoden. Im *Controller* implementiert der *Client-Stub* das Interface *IPerformance*. In der *Performance-Komponente* dagegen, implementiert und nutzt der *Server-Stub* das Interface *IPerformancer*.

Mit der Methode *startMeasure()* und *stopMeausure()* wird die Leistungsmessung auf der *Performance-Komponente* gestartet oder gestoppt. Über die Methode *getPerformanceContainer()* werden die gemessenen Leistungsmetriken an den *Controller* zurückgegeben.

3.4.2 Komponentenübergreifende Klassen

In diesem Abschnitt werden Klassen beschrieben, die komponentenübergreifend verwendet werden. Hierbei handelt es sich um Container-Klassen, die zur Übertragung der Messwerte zwischen den Komponenten übergeben werden.

Klassen WorkerContainer / WorkerValue

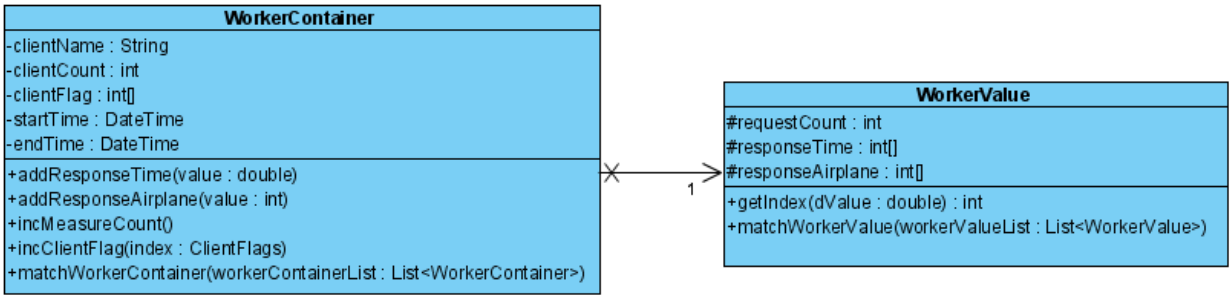


Abb. 3.4.2.1: UML-Klassendiagramm für die Klassen *WorkerContainer* und *WorkerValue*.

In den Klassen *WorkerContainer* und *WorkerValue* werden die von einem *Worker* ermittelten Messwerte während einer Simulation gespeichert. In der Klasse *WorkerContainer* werden allgemeine Informationen zur Simulation gespeichert, in der Klasse *WorkerValue* dagegen, die eigentlichen Messwerte. Jeder *WorkerContainer* enthält ein *WorkerValue-Objekt* und enthält somit auch die Messdaten. Nach dem Ende einer Simulation, wird von jedem *Worker* ein *WorkerContainer-Objekt* an den *Controller* zurückgegeben. In der Klasse *WorkerContainer* sind die nachfolgenden Attribute zur Messwertwertaufnahme enthalten:

Attribut (Messwert)	**Beschreibung**
clientName	Vom Controller zugewiesener Name zur Identifikation.
clientCount	Anzahl simulierter Benutzer
clientFlag	Verschiedene Kontrollflags zur Simulation der Benutzerlast wie z.B. Anzahl fehlerhafter Client-Anfragen.
startTime	Startzeitpunkt der Leistungsmessung
endTime	Endzeitpunkt der Leistungsmessung

Tabelle 3.4.2.1: Attribute (Kontrollflags) der Klasse *WorkerContainer*.

Das Attribut *clientName* dient zur Zuordnung des *WorkerContainers* zum jeweiligen *Worker*. Die Attribute *clientCount*, *clientFlag, startTime, endTime* und *requestCount* werden zu Kontrollzwecken der Simulation der Client-Anfragen gespeichert (s. Tab. 3.4.2.1). Im Attribut *clientFlag* werden verschiedene Zusatzinformationen über den Ablauf der Simulation gespeichert, um die Nachvollziehbarkeit des Verhaltens einer Simulation zu erhöhen (Transparenz). So werden folgende Indikatoren gespeichert:

Attribut (Messwert)	**Beschreibung**
reqCount	Anzahl durchgeführter Client-Anfragen mit und ohne Messwertaufnahme
reqFailCount	Anzahl fehlerhafter Client-Anfragen (Allgemeine Exception)
reqFailWebCount	Anzahl HTTP-Timeout einer Client-Anfrage (WebException)
noAirplane	Anzahl Client-Anfragen ohne Rückgabe von Flugzeugnachrichten

Tabelle 3.4.2.2: Fälle, die das clientFlag abspeichert.

In der Klasse *WorkerValue* dagegen, sind die nachfolgenden Attribute enthalten:

Attribut (Messwert)	**Beschreibung**
requestCount	Anzahl durchgeführter Client-Anfragen mit einer Messwertaufnahme.
responseTime	Kategorisierte Antwortzeiten aller Anfragen (eine Antwortzeit pro Anfrage)
responseAirplane	Kategorisierte Anzahl erhaltener Flugzeugnachrichten pro Anfrage (Client-Durchsatz).

Tabelle 3.4.2.3: Attribute (Messwerte) der Klasse *WorkerValue*.

Im Attribut *responseTime* werden die Antwortzeiten aller Client-Anfragen gespeichert (eine Antwortzeit pro Anfrage). Im Attribut *responseAirplane* dagegen, werden die Flugzeugnachrichten pro Anfrage gespeichert. Beide Messwerte werden kategorisiert abgespeichert. Die Kategorisierung erfolgt wie in Abschnitt 3.1 in der Anforderung 3 beschrieben. Dabei entspricht eine Kategorie einer Indexposition in den beiden Listen Attributen, siehe Abb. 3.4.2.2.

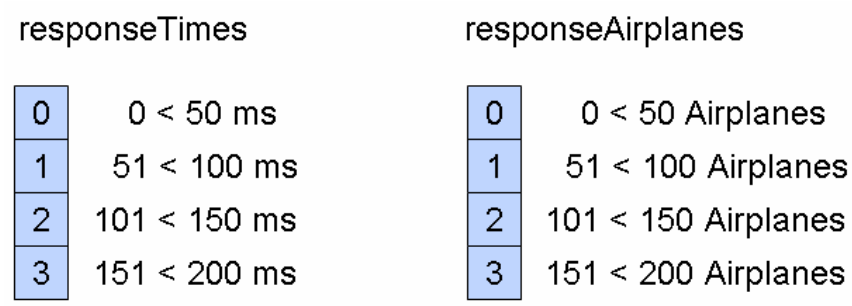


Abb. 3.4.2.2: Kategorisierung der Messwerte.

Zusammenfassend werden die Methoden beider Klassen beschrieben, um den Zugriff vom *WorkerContainer* auf das *WorkerValue-Objekt* zu verdeutlichen.

Mit den Methoden aus der Klasse *WorkerContainer addResponseTime(), addResponseAirplane(), incMeasureCount()* und *incClientFlags()* werden die verschiedenen Messwerte des *Workers* an den *WorkerContainer* zur Speicherung übergeben. Der *WorkerContainer* wiederum speichert die Messwerte in den entsprechenden Attributen ab. Mit der Methode *getIndex()* wird die ermittelte Antwortzeit und die Anzahl erhaltener Flugzeugnachrichten jeder Client-Anfrage kategorisiert. Die Methode liefert zu einem Messwert den entsprechenden Kategorisierungsindex, siehe Abb. 3.4.2.2.

In den beiden Klassen steht jeweils eine Methode zur Verfügung (*matchWorkerContainer()* oder *matchWorkerValue()*), die jeweils eine Zusammenfassung von Objekten bzw. den darin enthaltenen Attributen ermöglicht. Diese Funktionalität lässt sich mit der Akkumulatorfunktion einer Liste im .NET-Framework, die sich vom Interface *Enumerable* ableitet, vergleichen. In diesem Kontext wird einem neu erstellten Objekt (leeres Objekt) eine Liste von Objekten desselben Typs übergeben und die darin enthalten Attribute in dieses neue Objekt kopiert und zusammengefasst (akkumuliert). Diese Funktionalität wird vom *Worker* und vom *Controller* für die Zusammenfassung (Akkumulierung) der *WorkerContainer*- und WorkerValue-*Objekte* verwendet.

Klassen PerformanceContainer / PerformanceValue

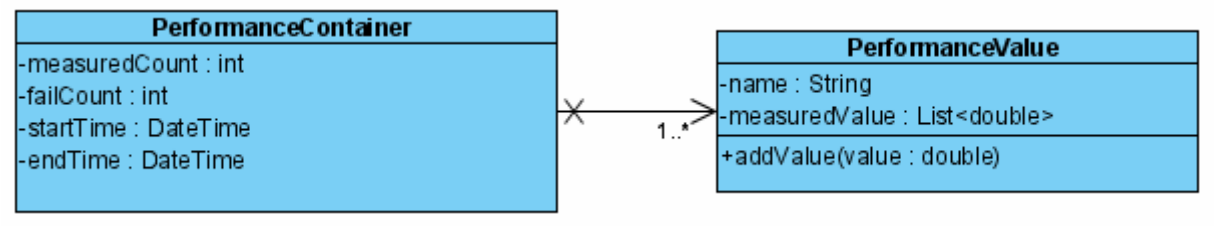


Abb. 3.4.2.3: UML-Klassendiagramm der Klassen *PerformanceContainer* und *PerformanceValue*.

In den Klassen *PerformanceContainer* und der *PerformanceValue* werden die Messwerte der Leistungsmetriken, die durch die *Performance-Komponente* ermittelt werden, gespeichert. Die Klasse *PerformanceContainer* enthält, wie der zuvor erwähnte *WorkerContainer*, allgemeine Informationen zur Leistungsmessung. Die Klasse *PerformanceValue* dagegen, repräsentiert die Messwerte einer bestimmten Leistungsmetrik. Der *PerformanceContainer* hält für jede zu messende Leistungsmetrik ein entsprechendes PerformanceValue-Objekt. Der Zugriff auf die Attribute der Klasse *PerformanceContainer* erfolgt mit Properties. Die gemessenen Werte einer Leistungsmetrik dagegen, werden über die Methode *addValue()* in der *PerformanceKlasse* in der Liste *measuredValue* abgelegt. Nach dem Ende eines Analyselaufes, wird das *PerformanceContainer-Objekt* an den *Controller* zurückgegeben.

Die Klasse *PerformanceContainer* enthält die nachfolgenden Attribute:

Attribut	**Beschreibung**
measuredCount	Anzahl durchgeführter Messungen
failCount	Anzahl fehlgeschlagener Messungen
startTime	Startzeitpunkt der Leistungsmessung
endTime	Endzeitpunkt der Leistungsmessung

Tabelle 3.4.2.4: Attribute der Klasse *PerformanceContainer*.

Die Klasse *PerformanceValue* enthält die nachfolgenden Attribute:

Attribut	**Beschreibung**
name	Name der Leistungsmetrik (z.B. CPU-Auslastung)
measuredValue	Enthält die Messwerte einer Leistungsmetrik

Tabelle 3.4.2.5: Attribute der Klasse *PerformanceValue*.

3.4.3 Controller-Komponente

Nachfolgend ein Auszug aus dem UML-Klassendiagramm der Controller-Komponente zur Realisierung der Funktionalität für die Analysesteuerung.

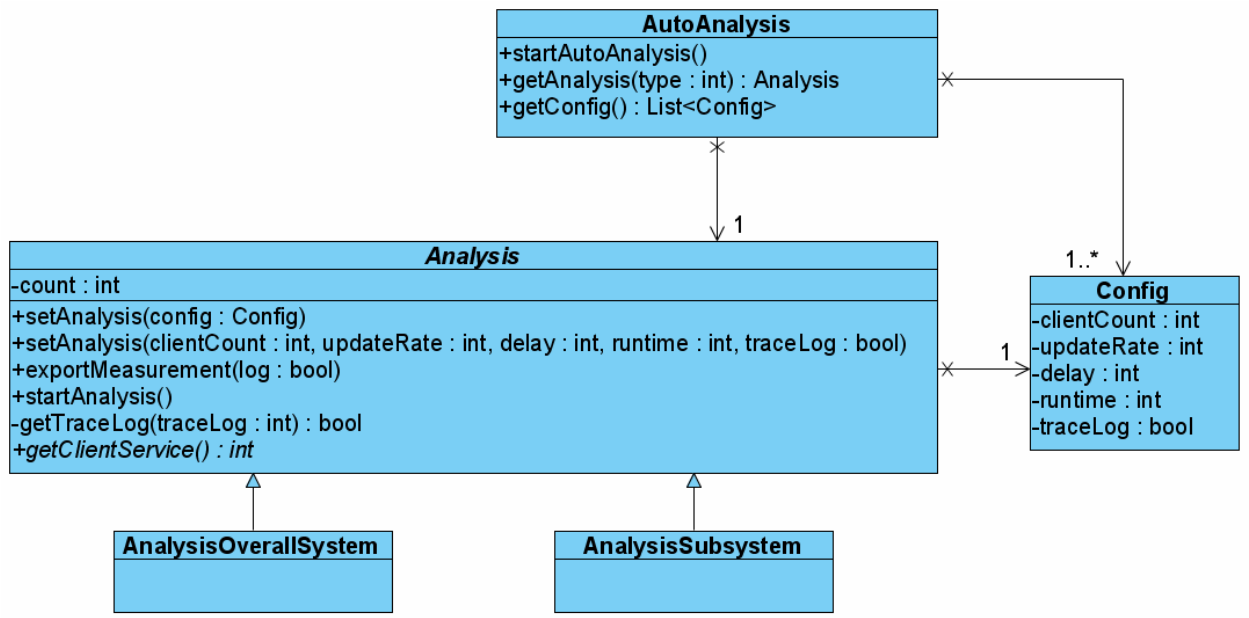


Abbildung 3.4.3.1: UML-Klassendiagramm zur Funktionalität der Analysesteuerung.

Klassen Analysis / AnalysisOverallSystem / AnalysisSubsystem

Bei der Klasse *Analysis* handelt es sich um eine abstrakte Klasse. Die Klassen *AnalysisOverallSystem* und *AnalysisSubsystem* werden von dieser abstrakten Klasse abgeleitet. Die beiden Klassen unterscheiden sich nur in der Implementierung der Methode *getClientService()*. Diese Methode liefert in Abhängigkeit des ausgewählten Analysemodus (*Gesamtsystem* oder *Teilsystem*) und der Delay-Einstellung (*mit oder ohne Delay*) eine Kennzahl zur späteren Ermittlung der Kommunikationsschnittstelle (*Web-Service* oder *TcpChannel*) im *Worker*.

Grundsätzlich ist die Klasse *Analysis* verantwortlich für die Steuerung der gesamten verteilten Anwendung zur Leistungsanalyse. Dies beinhaltet das Einlesen der Konfiguration für einen Analysedurchlauf, das Initiieren des Analysedurchlaufes durch Benachrichtigung der Worker- und der Performance-Komponente sowie das Beenden eines Analyselaufes. Nach dem Ende eines Analyselaufes werden die Messwerte eingesammelt und exportiert (CSV-Datei).

Das Einlesen der Konfigurationsparameter für einen Analyselauf, erfolgt entweder durch eine manuelle Eingabe in der Konsole oder automatisch durch die Klasse *AutoAnalysis*. Für beide Möglichkeiten steht jeweils eine Methode *setSimulation()* zur Verfügung. Ist die Konfiguration für einen Analyselauf übergeben, wird dieser über die Methode *startAnalysis()* gestartet. Diese Methode realisiert den gesamten Ablauf eines Analyselaufes. Dieser Ablauf kann aus den Abbildungen 3.3.2.1 und 3.4.3.2 entnommen werden.

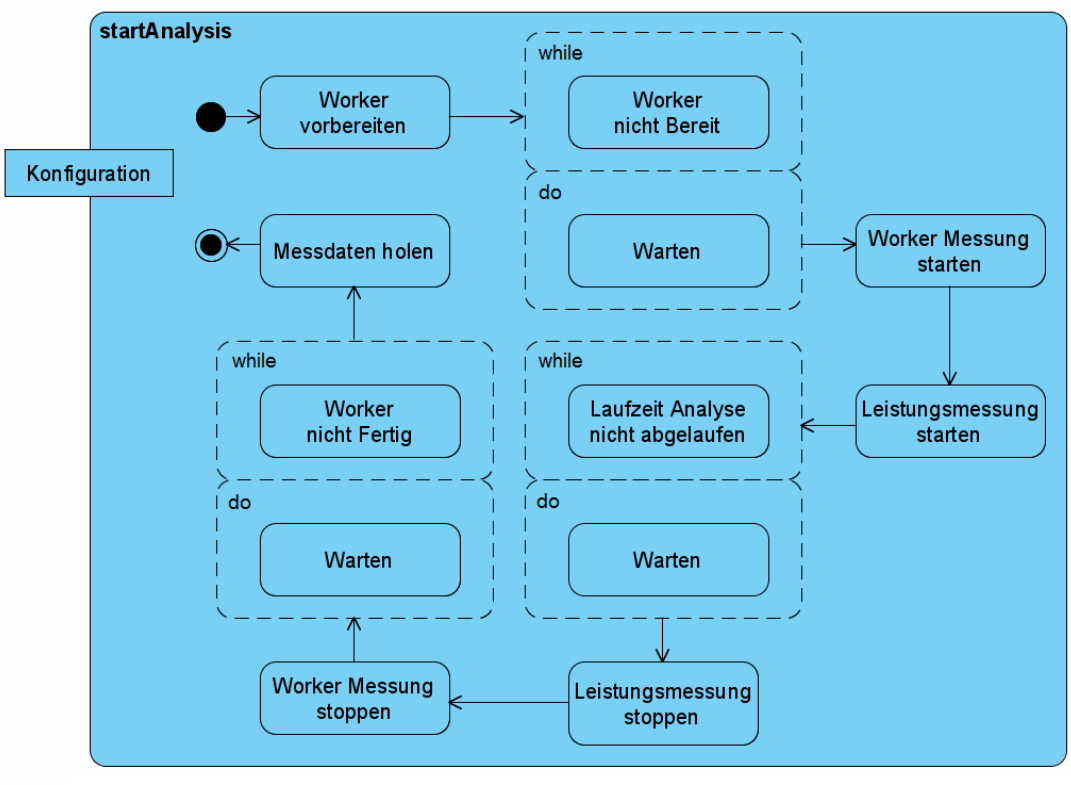


Abb. 3.4.3.2: UML-Aktivitätsdiagramm zum Ablauf der Methode *startAnalysis()*.

Klasse AutoAnalysis

Mit der Klasse *AutoAnalysis* wird eine automatische Durchführung von beliebig vielen Analyseläufen realisiert. Eine Eingabe einer Konfiguration über die Konsole ist somit nicht nötig. Die Konfigurationen für beliebig viele Analyseläufe werden über eine XML-Datei konfiguriert und eingelesen. Im Listing 3.4.3.1 ist ein Beispiel für die Konfiguration von drei Analyseläufen abgebildet. Alle eingelesenen konfigurierten Analyseläufe werden mit dem Aufruf der in Abbildung 3.4.3.2 beschriebenen Methode *startAnalysis()* sequentiell nacheinander ausgeführt. Dem voraus geht jedoch der Aufruf der Methode *setSimulation()* zur jeweiligen Übergabe der Konfiguration des Analyselaufes. Nach jedem Lauf erfolgt zudem ein automatischer Export der gemessenen Leistungsmetriken (CSV-Datei).

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <Config>
      <clientCount>50</clientCount>
      <updateRate>1000</updateRate>
      <delay>1</delay>
      <runtime>3</runtime>
      <traceLog>false</traceLog>
   </Config>
   <Config>
      <clientCount>100</clientCount>
      <updateRate>5000</updateRate>
      <delay>1</delay>
      <runtime>3</runtime>
      <traceLog>false</traceLog>
   </Config>
   <Config>
      <clientCount>150</clientCount>
      <updateRate>10000</updateRate>
      <delay>1</delay>
      <runtime>3</runtime>
      <traceLog>false</traceLog>
   </Config>
</ArrayOfConfig>
```

Listing 3.4.3.1: XML-Datei mit Konfigurationen für die automatische Analyse.

Klasse Config

In der Klasse *Config* hält die Klasse *Analysis* zentral die aktuelle Konfiguration eines Analyselaufes. Vor jedem Analyselauf wird das *Config-Objekt* mit den entsprechenden Konfigurationsparametern aktualisiert (überschrieben).

Attribut	**Beschreibung**
clientCount	Gesamtanzahl zu simulierenden Benutzer.
updateRate	Intervall in dem die Client-Anfragen erfolgen.
delay	Simulation der Benutzerlast mit oder ohne Delay (Standard- oder Privilegierter-Web-Client)
runtime	Laufzeit des Analyselaufes
tracelog	Kennzeichen für den Logging-Mechanismus (ja oder nein)

Tabelle 3.4.3.1: Attribute (Parameter) einer Konfiguration.

Nachfolgend ein Auszug aus dem UML-Klassendiagramm der *Controller-Komponente* zur Realisierung der Kommunikation dem *Worker*.

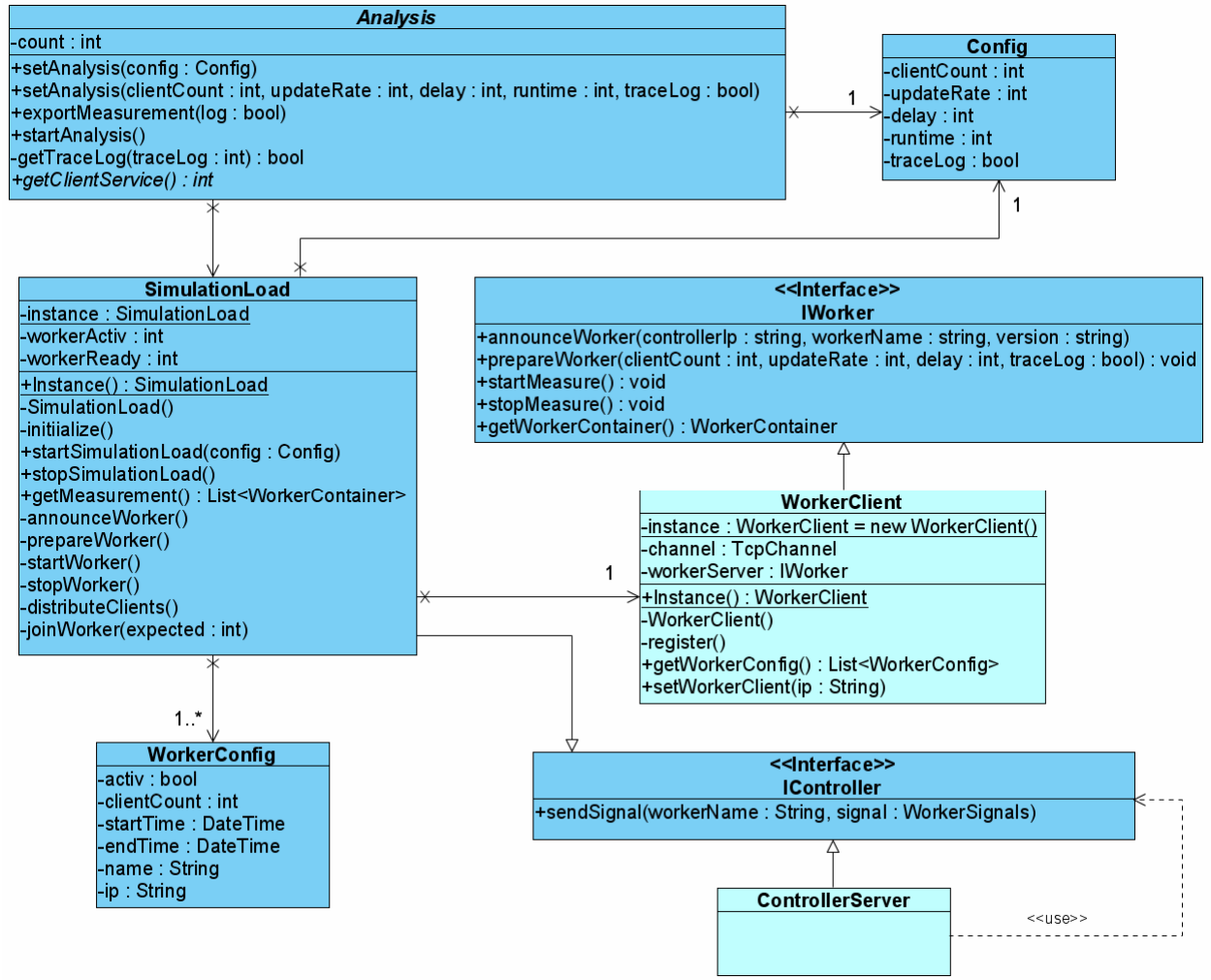


Abb. 3.4.3.3: UML-Klassendiagramm Auszug zur Kommunikation mit der Worker-Komponente.

Klasse SimulationLoad

Die Klasse *SimulationLoad* ist verantwortlich für die Steuerung der *Simulation der Benutzerlast* und kommuniziert hierfür über den *WorkerClient* (*Client-Stub*) mit den *Workern* über RPC-Aufrufe (Remote Procedure Calls). Für diese Kommunikation verwendet die Klasse *SimulationLoad* die deserialisierten (XML-Datei) Kommunikationsparameter aus der Klasse *WorkerConfig*, siehe Listing 3.4.3.2. Die *WorkerConfig-Objekte* (Kommunikationsparameter) erhält die Klasse *SimulationLoad* mit dem Aufruf der Methode *getWorkerConfig()* von dem *WorkerClient*. Die Konfiguration des aktuellen Analyselaufes erhält die Klasse *SimulationLoad* über eine Referenz zu einem *Config-Objekt.* Die Klasse *SimulationLoad* realisiert zudem über

das Interface *IController* die Funktionalität für die eingehende Kommunikation von einem *Worker*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ArrayOfWorkerConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-
3  instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4     <WorkerConfig>
5        <name>PC-1</name>
6        <ip>141.79.52.20</ip>
7     </WorkerConfig>
8     <WorkerConfig>
9        <name>PC-2</name>
10       <ip>141.79.52.21</ip>
11    </WorkerConfig>
12    <WorkerConfig>
13       <name>PC-3</name>
14       <ip>141.79.52.22</ip>
15    </WorkerConfig>
16 </ArrayOfWorkerConfig>
```

Listing 3.4.3.2: XML-Datei für die Konfiguration der *Worker.*

Die Methode *startAnalysis()* aus einer der beiden konkreten Klassen *OverallAnalysis* oder *SubsystemAnalysis* ruft die Methode *startLoadSimulation()* auf, um die Vorbereitungsphase der *Worker* einzuleiten. Die Methode *startLoadSimulation()* nutzt dafür mehrere private Methoden, die einzelne Unteraufgaben realisieren. In ihr werden nachfolgende Aktionen ausgeführt:

- Zuteilung der zu simulierenden Benutzer auf die *Worker*

Zuerst erfolgt eine Berechnung der Zuteilung der Benutzerlast auf die einzelnen verfügbaren *Worker* mit Hilfe der Methode *distributeClients().* Die einem *Worker* zugeteilte Menge an zu simulierenden Benutzer, wird im entsprechenden *WorkerConfig-Objekt* (Konfiguration eines Workers) im Attribute *clientCount* hinterlegt.

- Vorbereitung auf den *Workern* einleiten

Danach erfolgt der Aufruf der privaten Methode *prepareWorker()*. In dieser Methode wird über die Liste mit den *WorkerConfig-Objekten* iteriert und die darin enthaltenen IP-Adressen dazu verwendet, um über den *WorkerClient* (Client-Stub) die benötigten RPC-Aufrufe auf die *Worker* sequentiell ausführen zu können. Hierbei wird die gleichnamige RPC-Methode *prepareWorker()* auf jedem *Worker* ausgeführt. Damit wird die Vorbereitungsphase auf dem *Worker* selbst eingeleitet.

- Wartezustand (Synchronisation der *Worker*)

Dabei handelt es sich um den in Kapitel 3.3.2 angesprochenen Synchronisationsmechanismus Nach dem Einleiten der Vorbereitungsphase der *Worker*, wird die Methode *startLoadSimulation()* blockiert, womit der *Controller* in einen Wartezustand versetzt wird. Der Wartezustand wird erst wieder aufgehoben, so bald alle *Worker* ein *Ready-Signal* gesendet haben. Das Senden eines Ready-Signals erfolgt über einen RPC-Aufruf der Methode *sendSignal()* vom *Worker* zurück zum Controller. Bei der Ankunft eines Ready-Singnals wird das Attribut *workerReady* jedes Mal um 1 inkrementiert. Das aktive Warten selbst wird durch die Methode *joinWorker()* realisiert. Die Methode prüft zyklisch (Polling), ob der Wert des Attributes *WorkerReady* gleich der Anzahl der eingesetzten *Worker* entspricht. Die Methode *joinWorker()* wird also solange ausgeführt, bis alle *Worker* das Ready-Signal zurück gesendet haben. Ist dies der Fall terminiert die Methode *joinWorker()* und der Wartezustand wird verlassen.

- Start der Messwertaufnahme

Nach dem Ende des Wartezustandes wird auf den Workern die Messwertaufnahme gestartet. Hierfür erfolgt, wie zuvor beim Starten der Vorbereitungsphase, wiederum eine RPC-Aufruf der Methode *startMeasure().* Der Vorgang wird in der nachfolgenden Abbildung 3.4.3.4 dargestellt.

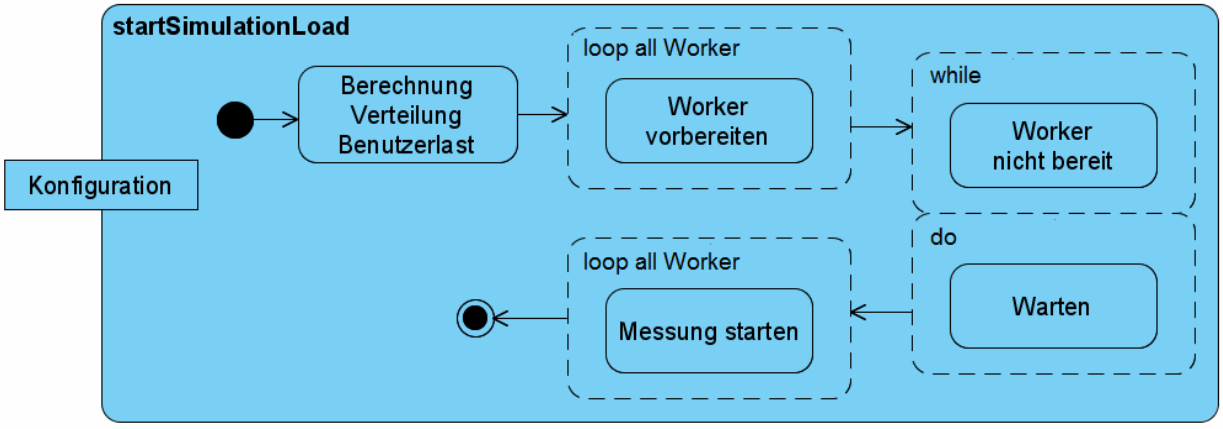


Abbildung 3.4.3.4: UML-Aktivitätsdiagramm zur Methode *startLoadSimulation().*

Die Methode *startAnalysis()* ruft die Methode stopLoadSimulation() nach Ende der Laufzeit eines Analyselaufes auf. Mit der Methode *stopLoadSimulation()* wird die *Simulation der Benutzerlast* auf den *Workern* wieder gestoppt. In der Methode *stopLoadSimulation()* erfolgt hierfür der Aufruf der privaten *stopWorker()* Methode. In dieser Methode erfolgen analog, wie beim Starten der Messwertaufnahme, RPC-Aufrufe der Methode *stopMeasure()* sequentiell für alle arbeitenden *Worker*, um das Beenden der Messwertaufnahme und die Simulation einzuleiten. Nach dem Einleiten der Beendigung der Simulation und der Messwertaufnahme auf den *Workern*, wird die Methode *stopLoadSimulation()* blockiert und der Controller in einen Wartezustand versetzt. Hier wird der Wartezustand erst wieder aufgehoben, sobald alle *Worker* das Beenden abgeschlossen und ein Finish-Signal gesendet haben. Bei der Ankunft eines Finish-Signals wird das Attribut *workerReady* (momentaner Wert = eingesetzte Worker) jedes Mal um 1 dekrementiert. Das aktive Warten selbst wird wieder durch die Methode *joinWorker()* realisiert. Die Methode prüft zyklisch (Polling), ob der Wert des Attributes *workerReady* nun wieder 0 ist. Ist dies der Fall terminiert die Methode *joinWorker()* und der Wartezustand wird verlassen. Dieser Ablauf wird in der nachfolgenden Abbildung 3.4.3.5 dargestellt.

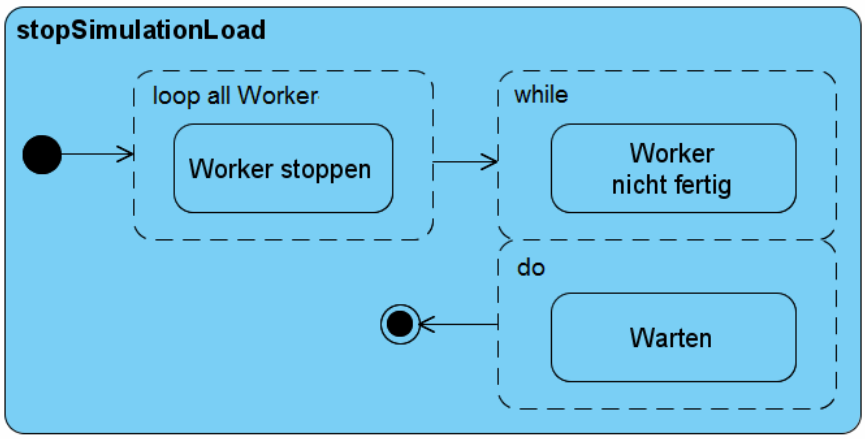


Abbildung 3.4.3.5: UML-Aktivitätsdiagramm zur Methode *stopSimulationLoad()*.

Mit der Methode *getMeasurement()* werden die gemessenen Leistungsmetriken der *Worker* eingesammelt. Diese wird nach der Beendigung der Simulation und der Messwertaufnahme ebenfalls durch die Methode *startAnalysis()* aufgerufen. Wie beim Start- oder Stoppvorgang der Worker erfolgt auch hier wieder ein RPC-Aufruf an jeden Worker. Dafür wird die Methode *getWorkerContainer()* aus dem *WorkerClient* (Client-Stub) aufgerufen. Jeder *Worker* liefert dabei eine *WorkerContainer-Objekt* in dem Messwerte gespeichert sind zurück. Dieser Ablauf wird in der nachfolgenden Abbildung 3.4.3.6 dargestellt.

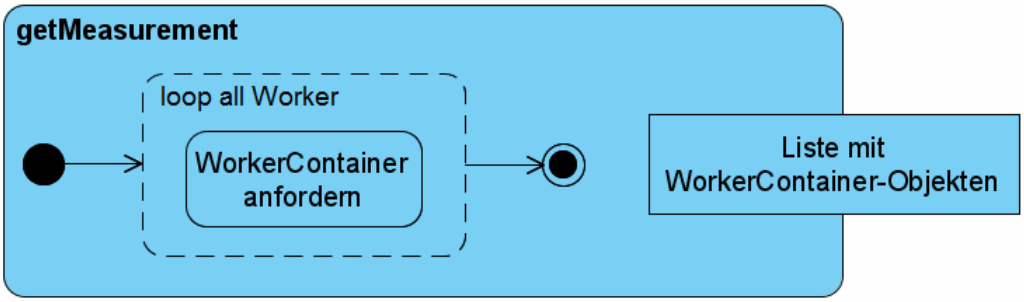


Abbildung 3.4.3.6: UML-Aktivitätsdiagramm zur Methode *getMeasurement().*

Klasse WorkerConfig

In der Klasse *WorkerConfig* werden unter anderem die zuvor eingelesenen Konfigurationen der Worker gehalten (siehe Listing 3.4.3.2). Die Klasse enthält die nachfolgenden Attribute:

Attribut	**Beschreibung**
activ	Kennzeichnet, dass der Worker für einen Analyselauf verwendet wird.
clientCount	Zugeteilte Anzahl zu simulierende Benutzer.
startTime	Startzeitpunkt der Messwertaufnahme auf dem Worker für den aktuellen Analyselauf
endTime	Endzeitpunkt der Messwertaufnahme auf dem Worker für den aktuellen Analyselauf
Ip	IP-Adresse des Worker die für die Kommunikation benötigt wird.
name	Name (Alias) des Worker zur Identifikation.

Tabelle 3.4.3.2: Attribute der Klasse WorkerConfig.

Der *clientCount* gibt an wie viele Benutzer dem *Worker* zugeteilt werden. Das Attribut *activ* kennzeichnet einen *Worker* für den aktuellen Analyselauf als aktiviert bzw. deaktivert. Wird

beispielsweise eine geringere Menge an Benutzer simuliert als Worker zur Verfügung stehen, dann werden nicht alle *Worker* benötigt und für den aktuellen Analyselauf vorübergehend deaktiviert.

Um die Nachvollziehbarkeit der Leistungsmessung zu erhöhen werden die Start- und Endzeitpunkte der Messwertaufnahme auf den Workern in den Attributen *startTime* bzw. *endTime* im jeweiligen WorkerConfig-Objekt bis zur Messwertrückgabe gespeichert. Mit dem Erhalt der WorkerContainer-Objekte im *Controller* werden diese Zeiten dann in dem jeweiligen WorkerContainer-Objekt eines jeweiligen *Workers* übernommen. Die Werte werden nur deshalb auf dem Controller aufgenommen und gespeichert, da in einem verteilten System die Synchronität der Systemzeiten auf den einzelnen Clients nicht vorausgesetzt werden kann.

Klassen WorkerClient / ControllerServer

Bei der Klasse *WorkerClient* handelt es sich um den *Client-Stub* der für die Kommunikation vom *Controller* zu einem *Worker* benötigt wird. Für die Realisierung des *Client-Stubs*, implementiert die Klasse *WorkerClient* das in Kapitel 3.4.1 beschriebene Interface *IWorker*. Die Klasse *WorkerClient* (Client-Stub) wird von der Klasse *SimulationLoad* genutzt.

Bei der Klasse *ControllerServer* handelt es sich um den *Server-Stub*, der für die eingehende Kommunikation von mit einem *Worker* benötigt wird. Der *Server-Stub* implementiert und nutzt hierfür das in Kapitel 3.4.1 beschriebene Interface *IController*. Die Funktionalität der *ControllerServer* wird von der Klasse *SimulationLoad* realisiert.

Nachfolgend ein Ausschnitt aus dem UML-Klassendiagramm der Controller-Komponente zur Realisierung der Kommunikation mit der Performance-Komponente auf dem Flugdatenserver.

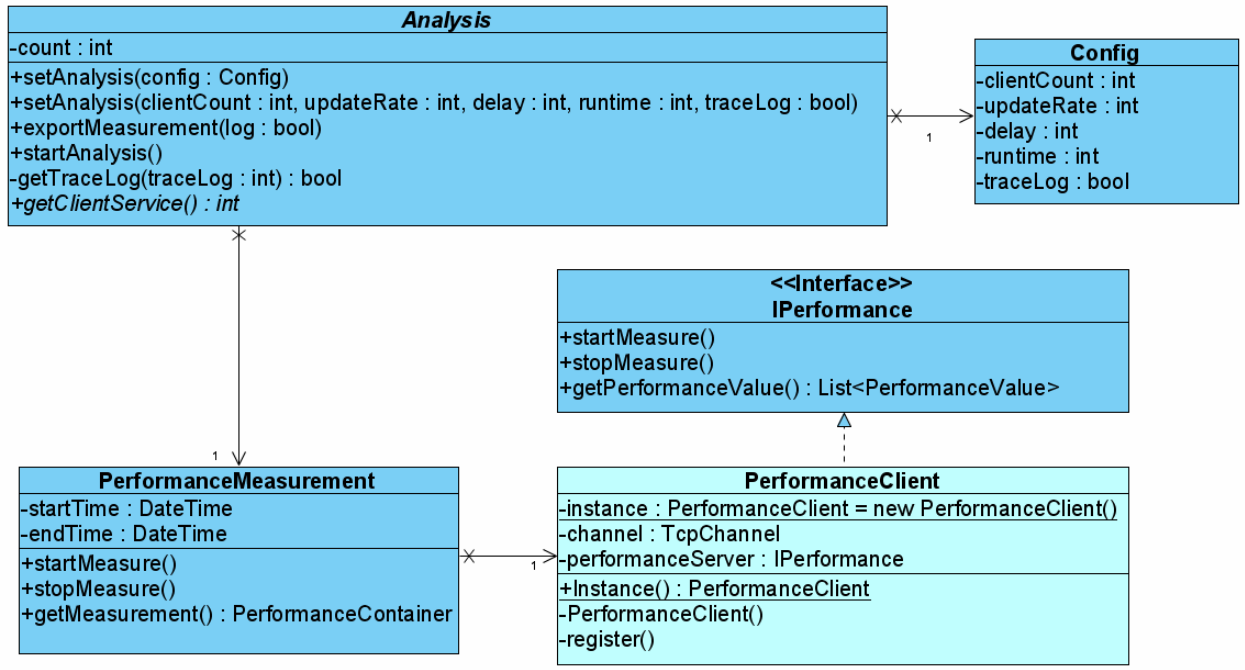


Abb. 3.4.3.7: UML-Klassendiagramm Ausschnitt zur Kommunikation mit der *Performance-Komponente*.

Klasse PerformanceMeasurement

Die Klasse *PerformanceMeasurement* ist verantwortlich für die Steuerung der Leistungsmessung während eines Analyselaufes. Hierfür kommuniziert die Klasse *PerformanceMeasurement* mit der *Performance-Komponente* und führt die entsprechenden RPC-Aufrufe aus. Zur Realisierung der Kommunikation mit der Performance-Komponente nutzt die Klasse *PerformanceMeasurement* den *PerformanceClient* (Client-Stub).

Mit den Methoden *startMeasure()* bzw. *stopMeasure()* wird die Leistungsmessung auf der Performance-Komponente gestartet bzw. gestoppt. Mit der Methode *getMeasurement()* erfolgt die Übergabe der gemessenen Leistungsmetriken in Form eines *PerformanceContainer-Objektes* (siehe Kapitel 3.4.2) an den Controller. Die drei Methoden werden von der Methode *startAnalysis()* aufgerufen.

Klasse PerformanceClient

Bei der Klasse *PeformanceClient* handelt es sich um den Client-Stub der für die Kommunikation mit der Performance-Komponente auf dem Flugdatenserver benötigt wird. Für die Realisierung des Client-Stubs implementiert der *PerformanceClient* das in Kapitel 3.4.1 beschriebene Interface *IPerformance*. Der *PerformacneClient* (Client-Stub) wird von der Klasse *PerformanceMeasurement* genutzt.

3.4.4 Worker-Komponente

Nachfolgend ein Auszug aus dem UML-Klassendiagramm der *Worker-Komponente* zur Realisierung der Basisfunktionalität.

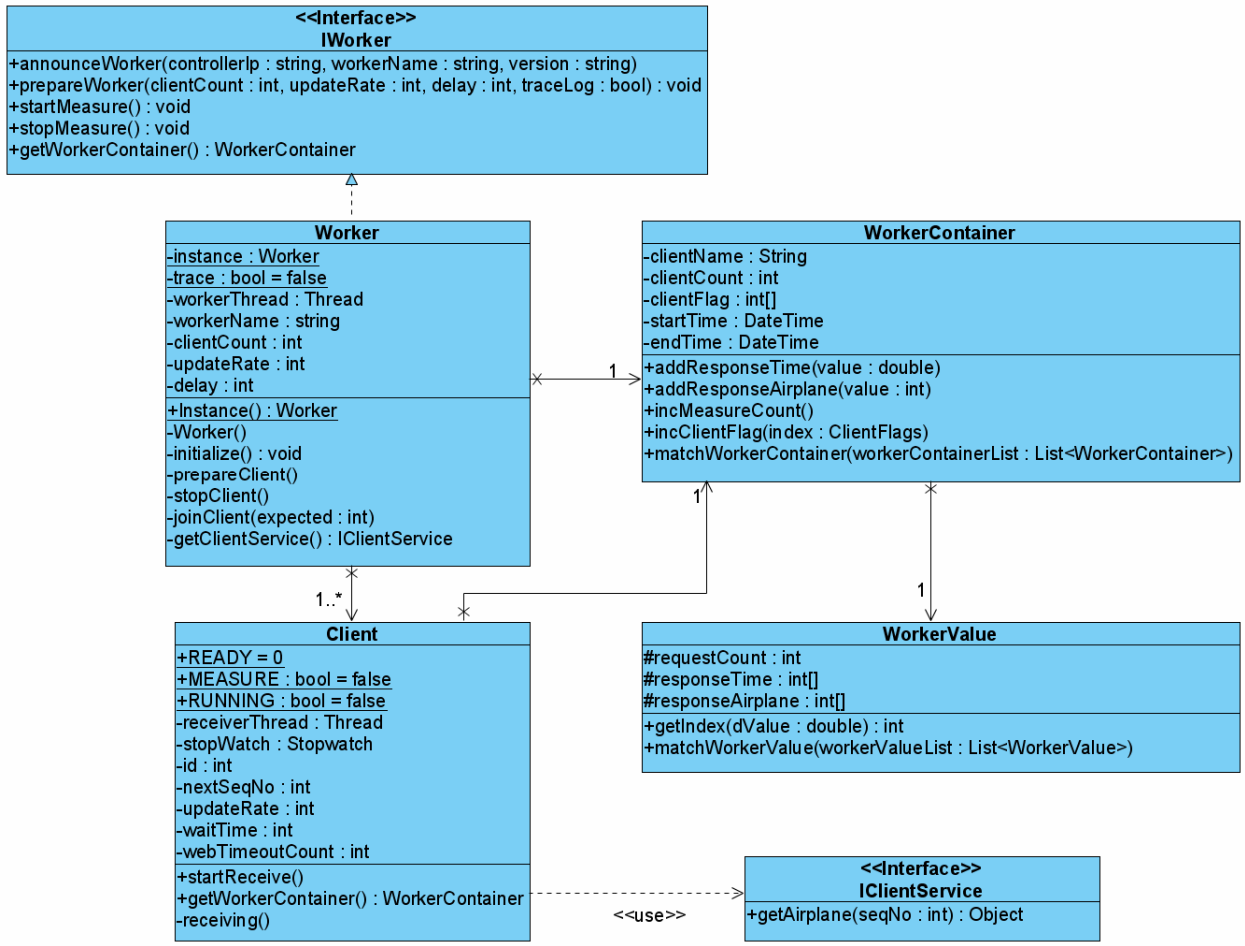


Abb. 3.4.4.1: UML-Klassendiagramm Basis-Klassen der *Worker-Komponente*.

Klasse Worker

Die Klasse *Worker* reagiert auf die RPC-Aufrufe des *Controllers* zur Steuerung der Simulation der Benutzerlast auf den Flugdatenserver. Nachfolgend wird der Ablauf der Simulation der Benutzlast über die RPC-Aufrufe beschrieben.

- Vorbereitung der Simulation der Benutzerlast (Wartezustand)

Zu Beginn ruft der Controller die Methode *prepareWorker() auf,* womit die Vorbereitung der Simulation auf dem *Worker* eingeleitet wird. Die Vorbereitung wird in dieser Methode jedoch nicht ausgeführt. Es wird lediglich ein neuer Thread erzeugt, in dem die eigentliche Vorbereitung dann tatsächlich durchgeführt wird. Der Thread wird mit der privaten Methode *prepareClient()* realisiert. Unmittelbar nach dem Erzeugen des neuen Threads terminiert der RPC-Aufruf der Methode *prepareWorker() im Controller.* Damit wird erreicht, dass die Vorbereitung nicht innerhalb eines RPC-Aufrufes erfolgt, sondern separat nur auf dem jeweiligen *Worker*, so dass der Controller unmittelbar nach der Terminierung die Vorbereitung auf weiteren *Workern* einleiten kann. Dadurch wird die Vorbereitung von den *Workern* parallel ausgeführt. Würde die Vorbereitung innerhalb eines RPC-Aufrufes erfolgen, hätte das zur Folge, dass die Vorbereitung auf den einzelnen *Worker* nacheinander (sequentiell) ausgeführt wird, was einen höheren Zeitaufwand bedeuten würde, als die zuvor erwähnte parallele Durchführung.

Die Vorbereitung wird dann durch die Thread-Methode *prepareClient()* tatsächlich realisiert. In der Thread-Methode wird für jeden zu simulierenden Benutzer ein Client-Objekt instanziiert. Mit der Instanziierung eines Client-Objektes wird eine konkrete Strategie für den Zugriff auf den Flugdatenserver erstellt. Diese konkrete Strategie erhält er über den Aufruf der Factory-Methode *getClientService(),* die diese konkrete Strategie über das Attribute *Delay* aus derselben Klasse ermittelt. Weiterhin wird mit der Instanziierung pro Client-Objekt ein Thread erzeugt und gestartet, der die Client-Anfragen an den Server stellt. Mit diesem Thread beginnt das Client-Objekt direkt Client-Anfragen an den Flugdatenserver zu stellen. Die Messwertaufnahme findet zu diesem Zeitpunkt noch nicht statt. Dieser Vorgang wird in Abbildung 3.4.4.2 dargestellt.

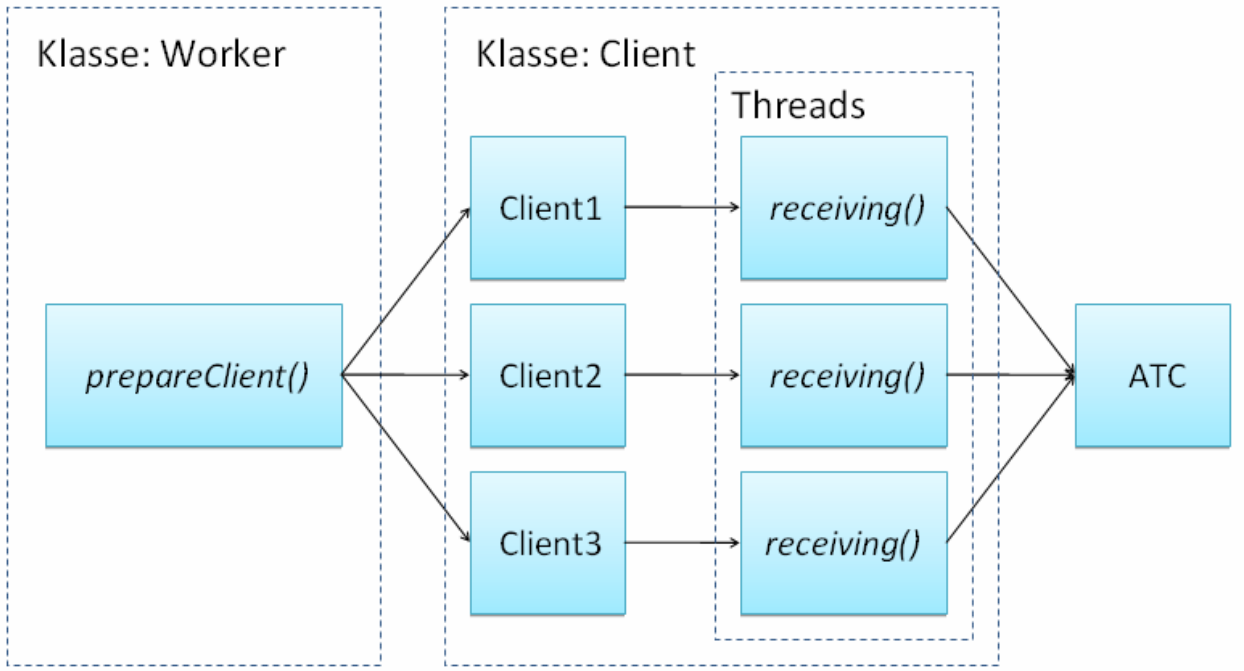


Abb. 3.4.4.2: Ablauf der Vorbereitung zur Client-Erstellung.

Nach dem Instanziieren aller *Client-Objekte,* befindet sich die Methode *prepareClient()* in einem Wartezustand. Dieser Wartezustand wird nach demselben Verfahren wie zuvor beim *Worker* allerdings mit der privaten Methode *joinClients()* erzeugt. Die Methode *prepareClient()* bleibt dabei solange im Wartezustand bis alle *Client-Objekte* Ihre erste Client-Anfrage erfolgreich ausgeführt und beantwortet bekommen haben. Danach wird der Wartezustand verlassen und über den *ControllerClient (Client-Stub)* ein *Ready-Signal* an den *Controller* gesendet. Damit ist der *Worker* bereit für das Starten der Messwertaufnahme. In der nachfolgenden Abbildung 3.4.4.3 wird dieser Ablauf noch einmal dargestellt.

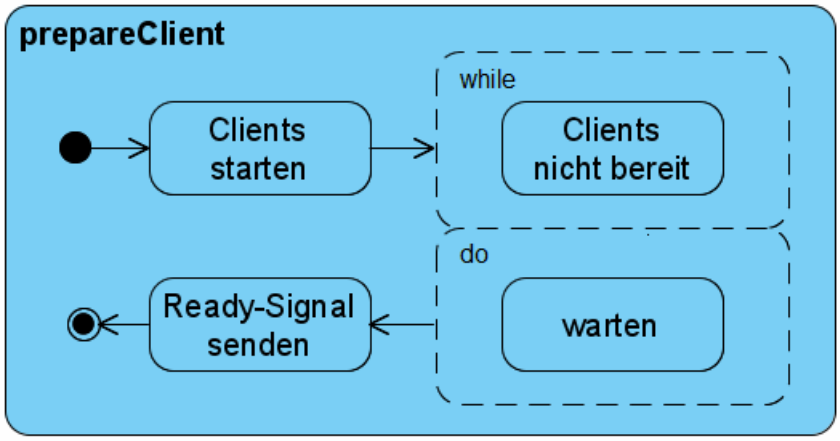


Abb. 3.4.4.3: UML-Aktivitätsdiagramm zur Methode *prepareClient().*

- Start der Messwertaufnahme

Mit dem RPC-Aufruf der Methode *startMeasure()* auf dem *Worker* durch den *Controller* wird das statische Attribut *MEASURE* auf *TRUE* gesetzt. Damit startet auf einem *Worker* ab diesem Zeitpunkt die Messwertaufnahme während der Durchführung der Client-Anfragen.

- Stopp der Messwertaufnahme und Wartezustand

Mit dem Aufruf der Methode *stopWorker()* durch den *Controller,* wird die Messwertaufnahme gestoppt und das Beenden der Simulation eingeleitet. Zuerst wird in dieser Methode das statische Attribut MEASURE auf FALSE gesetzt und damit ist die Messertaufnahme ab diesem Zeitpunkt gestoppt. Dann wird wie bei der zuvor erwähnten Vorbereitung auch, das Beenden der Simulation über einen separaten Thread durch die private Methode *stopClients()* eingeleitet. Bei der Beeindung der Simulation werden die einzelnen Threads beendet, indem auf die Terminierung jedes einzelne Threads gewartet wird. Dieser gesamte Wartevorgang kann sehr zeitintensiv werden. Die Methode *stopClient()* ist nun solange im Wartezustand, bis alle Threads der Client-Objekte terminiert sind. Ist das der Fall, sendet der *Worker* über den *ControllerClient* das Finish-Signal an den Controller. Hiermit ist die Simulation beendet, so dass der *Controller* die Messwerte anfordern kann.

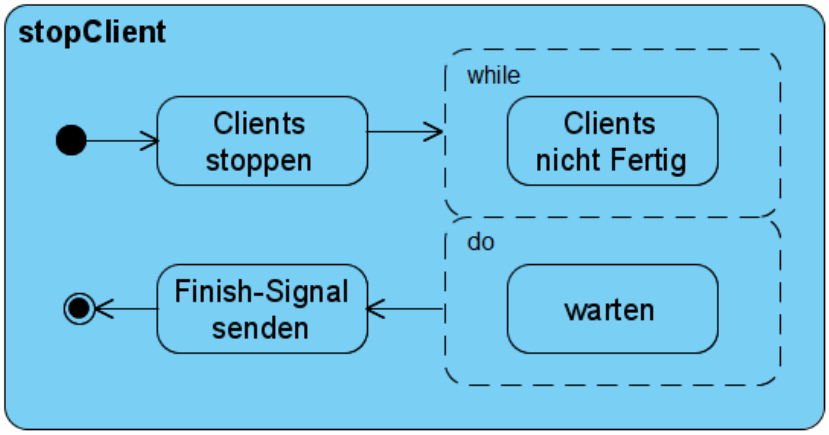


Abb. 3.4.4.4: UML-Aktivitätsdiagramm zur Methode *stopClient().*

- Rückgabe der Messwerte

Mit dem Aufruf der Methode *getMeasurement() durch den Controller,* sammelt die Klasse *Worker* die *WorkerContainer* jedes Client-Objektes ein. Danach werden diese eingesammelten *WorkerContainer* mit Hilfe der Methode *matchWorkerContainer()* zu einem neuen *WorkerContainer* zusammengefasst, welches dann an den Controller zurückgegeben wird.

Klasse Client

Die Klasse *Client* realisiert die Durchführung der Client-Anfragen zur Simulation der Benutzerlast. Für jeden simulierten Benutzer wird ein *Client*-Objekt erstellt. Jedes dieser *Client*-Objekte realisiert die Thread-Methode *receiving()*, die innerhalb eines vorgegeben Intervalls (Update-Rate) Client-Anfragen an den Flugdatenserver stellt. Die Client-Anfragen an den Flugdatenserver werden über das Interface *IClientService* ermöglicht. Weiterhin realisiert jedes Client-Objekt für sich die Messwertaufnahme. Die Messdaten werden dabei in einem *WorkerContainer* gespeichert. Jedes *Client-Objekt* hat sein eigenes *WorkerContainer*. Dies hat den Vorteil dass kein Synchronisationsmechanismus nötig ist, der bei einem gemeinsamen Objektzugriff auf das *WorkerContainer* nötig wäre. Die Klasse *Worker* führt am Ende einer Simulation die *WorkerContainer* aller Clients zu einem einzigen zusammen. Das Zusammenfassen der *WorkerContainer* der Clients zu einem *WorkerContainer* erfolgt mit der Methode *matchWorkerValue()* aus der *WorkerValue*-Klasse.

Nachfolgend wird der Ablauf der Thread-Methode *receiving()* mit einem UML-Sequenzdiagramm dargestellt. In dem nachfolgenden UML-Sequenzdiagramm wird zur Vereinfachung nur die Ermittlung und Speicherung der Antwortzeiten dargestellt. Die Ermittlung und Speicherung anderer Messwerte wie in Kapitel 3.4.2 beschrieben werden zur Vereinfachung nicht dargestellt.

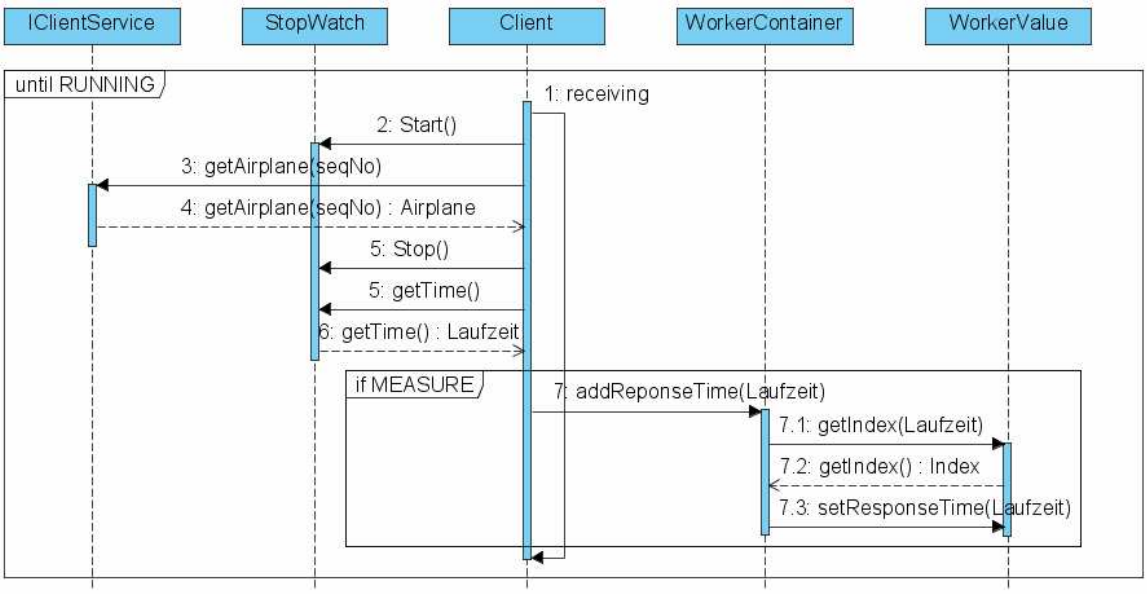


Abb. 3.4.4.5: UML-Sequenzdiagramm zur Simulation von Client-Anfragen.

In der Thread-Methode *receiving()* wird vor dem Stellen der Client-Anfrage die Zeitmessung mit der Methode *start()* der Klasse *StopWatch* gestartet. Unmittelbar danach erfolgt die eigentliche Client-Anfrage (Request) an den Flugdatenserver über eine konkrete Strategie des Interface *IClientService*. Hierfür ist die Übergabe der in Abschnitt 2.2.4 beschriebenen Sequenznummer nötig. Auf die Client-Anfrage erhält der *Worker* die Antwort des Servers (Response), die eine Menge an Flugzeugnachrichten beinhaltet. Unmittelbar nach Erhalt der Antwort, wird die Zeitmessung über die Methode *Stop()* der Klasse *StopWatch* gestoppt. Danach erfolgt die Abfrage der gemessenen Laufzeit über die Methode *getTime()* aus der Klasse *StopWatch*. Hat der *Worker* das Signal zur Leistungsmessung bereits erhalten, erfolgt die Speicherung der gemessenen Antwortzeit im *WorkerContainer* über die Methode *addResponseTime()*. Hierfür erfolgt für die kategorisierte Abspeicherung der Antwortzeiten die Ermittelung des Kategorie-Index. Ist der Kategorie-Index ermittelt, wird die Antwortzeit kategorisiert im *WorkerValue* gespeichert. Danach wird über den Zeitraum der Update-Rate gewartet (*Thread.Sleep()*). Der gesamte Vorgang wird solange wiederholt, bis das statische Attribut *RUNNING* durch den RPC-Aufruf der Methode *stopWorker()* durch den *Controller* im *Worker* auf *FALSE* gesetzt wird.

Nachfolgend ein Auszug aus dem UML-Klassendiagramm der Worker-Komponente zur Realisierung der Kommunikation mit dem Flugdatenserver.

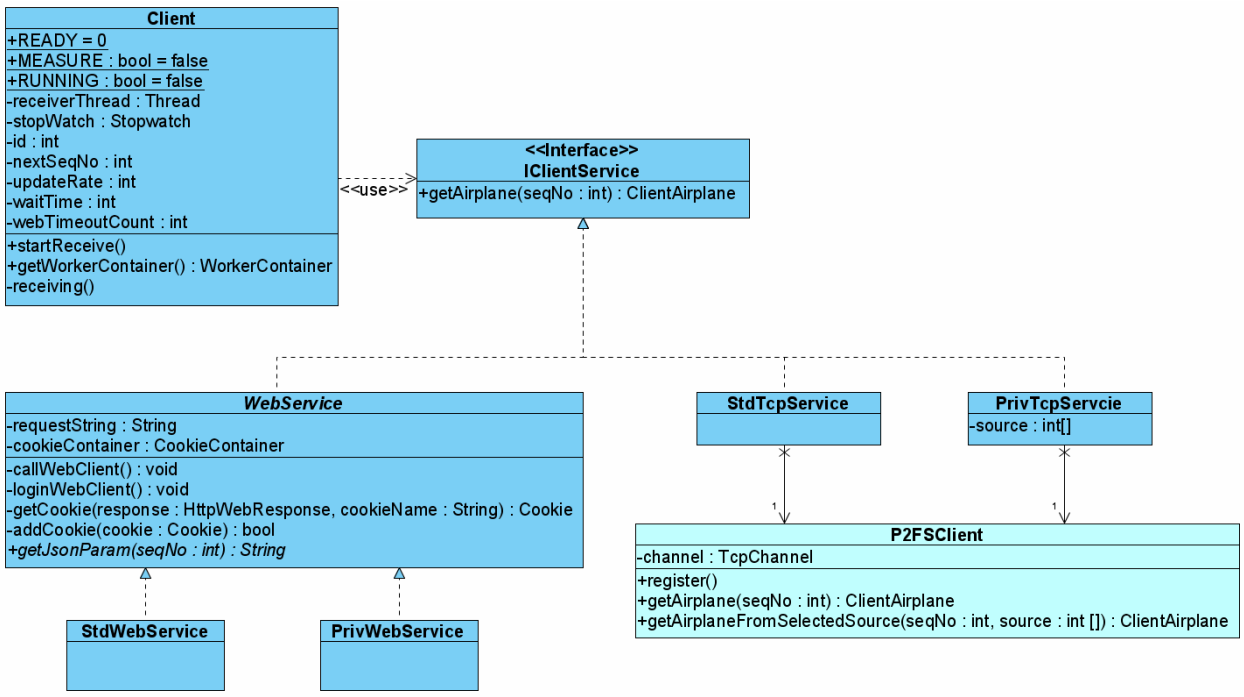


Abb. 3.4.4.6: UML-Klassendiagramm Auszug zur Kommunikation mit dem Flugdatenserver.

Interface IClientService

Das Interface *IClientService* realisiert die Schnittstelle für die Durchführung der Client-Anfragen an den Flugdatenserver. Hiefür wird das *Strategy Pattern* verwendet. Das Interface fordert die Implementierung der Methode *getAirplane()*. Diese Methode erwartet als Übergabeparameter eine Sequenznummer. Wie in Kapitel 2.2.4 beschrieben, erfolgt anhand dieser Sequenznummer die Bereitstellung der Flugzeugnachrichten (Airplane-Objekte) durch den *ATC.Server*. In Abhängigkeit des Analysemodus (*Gesamtsystem* oder *Teilsystem*) und der Delay-Einstellung (mit oder ohne Delay) wird die entsprechende konkrete Strategie zur Kommunikation mit dem Flugdatenservers verwendet.

Für eine Analyse des *Gesamtsystems,* stehen die beiden konkreten Strategien *StdWebService* und *PrivWebService* zur Verfügung. Beiden Klassen wiederum leiten sich von der abstrakten Klasse *WebService* ab. Für die Analyse des *Teilsystems* dagegen stehen die beiden Strategien *StdTcpService* und *PrivTcpService* zur Verfügung.

Klassen WebService / StdWebService / PrivWebService

Die abstrakte Klasse *WebService* bzw. die konkreten Klassen *StdWebService* und *PrivWebService* realisieren den Zugriff auf den Web-Service auf dem ATC.Webserver für die Durchführung der Client-Anfragen. Die Klasse *StdWebService* wird für Client-Anfragen *mit Delay* verwendet und nutzt dabei den *Standard-Web-Service*. Die Klasse *PrivWebService* dagegen wird für Client-Anfragen *ohne Delay verwendet und nutzt den Privilegierten-Web-Service.*

Der Zugriff auf den Web-Service erfolgt wie im Web-Client (siehe Kapitel 2.2.3) mit HTTP-Anfragen (REST-Ansatz), also nicht über das SOAP-Protokoll. Die URL für die HTTP-Anfrage ist in beiden konkreten Klassen im Attribut *requestString* unterschiedlich definiert. Zudem unterscheidet sich in den beiden Klassen die Implementierung der Methode *getJsonParam()*. Diese Methode liefert in Abhängigkeit der gewählten Strategie die Übergabeparameter im JSON-Datenformat für die auszuführende Web-Service-Anfrage. Die eigentliche Aktualisierungsanfrage über neue Flugzeugnachrichten wird über die Methode *getAirplane()* realisiert.

Um die *Simulation der Benutzerlast* möglichst realistisch zu gestalten, muss zuerst ein programmatisches Login erfolgen, wie es ein echter Benutzer über den Browser manuell durchführen würde. Damit der Aufruf der Login-Seite der Web-Anwendung und das Login selbst simuliert werden kann, ruft die Methode getAirplane() bei ihrem ersten Aufruf die beiden privaten Methoden *callWebClient()* und *loginWebClient()* auf. Nach den beiden Aufrufen erhält der simulierte Benutzer in der Server-Antwort verschiedene Cookies, wie z.B. Cookies für eine *Session Id* und zur Authentifizierung. Diese Cookies werden mit der privaten Methode *getCookie()* aus dem HTTP-Header der jeweiligen Server-Antwort gelesen und mit der privaten Methode *addCookie()* einem bekannten *CookieContainer* (Liste von Cookies) hinzugefügt. Für die nachfolgenden Client-Anfragen über die Methode *getAirplane()* werden die erhaltenen Cookies, die sich nun im *CookieContainer* befinden, immer mitgegeben. Damit wird die *Simulation der Benutzerlast* bzw. die Client-Anfragen zur Aktualisierung der Flugbewegungen, wie in der echten Flugdatenserver Web-Anwendung, ausgeführt.

Klassen StdTcpService / PrivTcpService / P2FSClient

Die Klasse *StdTcpService* wird für eine Simulation unter Berücksichtung des Delays (*Standard-Web-Client*) und die Klasse *PrivTcpServcie* für eine Simulation ohne Delay (*Privilegierter-Web-Client*) verwendet. Dabei benutzen beide die Klasse *P2FSClient*. Bei dieser handelt es sich um den *Client-Stub* zur Kommunikation über den *TcpChannel* zum *ATC.Server*. In dieser Klasse werden die beiden Methoden *getAirplane()* und *getAirplaneFromSelectedSource()* bereitgestellt. Die Methode *getAirplane()* für von der Klasse *StdTcpService* genutzt und realisiert die Bereitstellung der Flugzeugnachrichten mit Berücksichtigung des Delays. Die Methode *getAirplaneFromSelectedSource()* dagegen wird vom *PrivTcpService* genutzt und realisiert die Bereitstellung der Flugzeugnachrichten ohne Delay.

Bei der Klasse *P2FSClient* handelt es sich um den Client-Stub der die Kommunikation über den TcpChannel mit dem ATC.Server realisiert. Dabei handelt es sich um denselben Client-Stub, den auch der ATC.Webserver benutzt.

Nachfolgend ein Auszug aus dem UML-Klassendiagramm der Worker-Komponente zur Realisierung der bidirektionalen Kommunikation mit dem Controller.

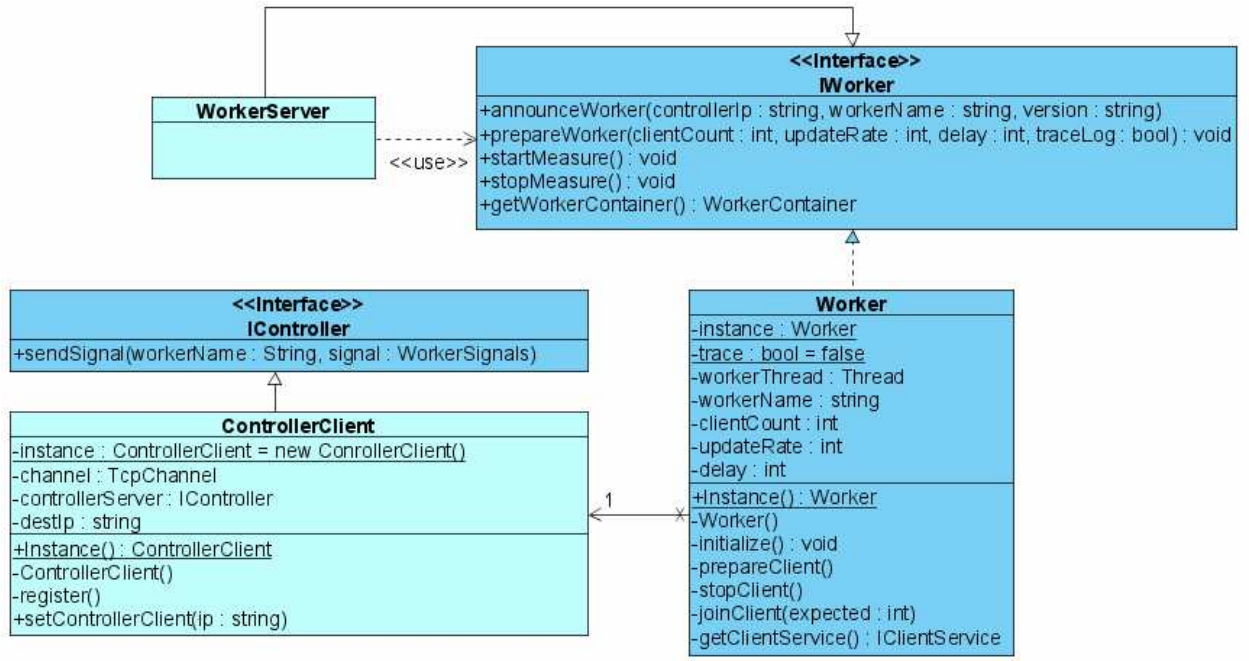


Abb. 3.4.4.7: UML-Klassendiagramm Auszug zum Client- und Server Stub.

Klassen ControllerClient / WorkerServer

Für die bidirektionale Kommunikation mit dem Controller implementiert der *Worker* einen *Client-Stub* und einem *Server-Stub*. Bei der Klasse *WorkerServer* handelt es sich um den *Server-Stub* für die Kommunikation vom Controller zum *Worker*. Der *ControllerClient* dagegen realisiert den *Client-Stub* zur Kommunikation in die Gegenrichtung vom *Worker* zum *Controller*. Für die Realisierung des Server-Stubs implementiert und nutzt die *WorkerServer* das Interface *IWorker*. Der *ControllerClient* dagegen implementiert das *Interface IController* und wird von der Klasse *Worker* genutzt.

3.4.5 Performance-Komponente

Nachfolgend das vollständige UML-Klassendiagramm der Performance-Komponente.

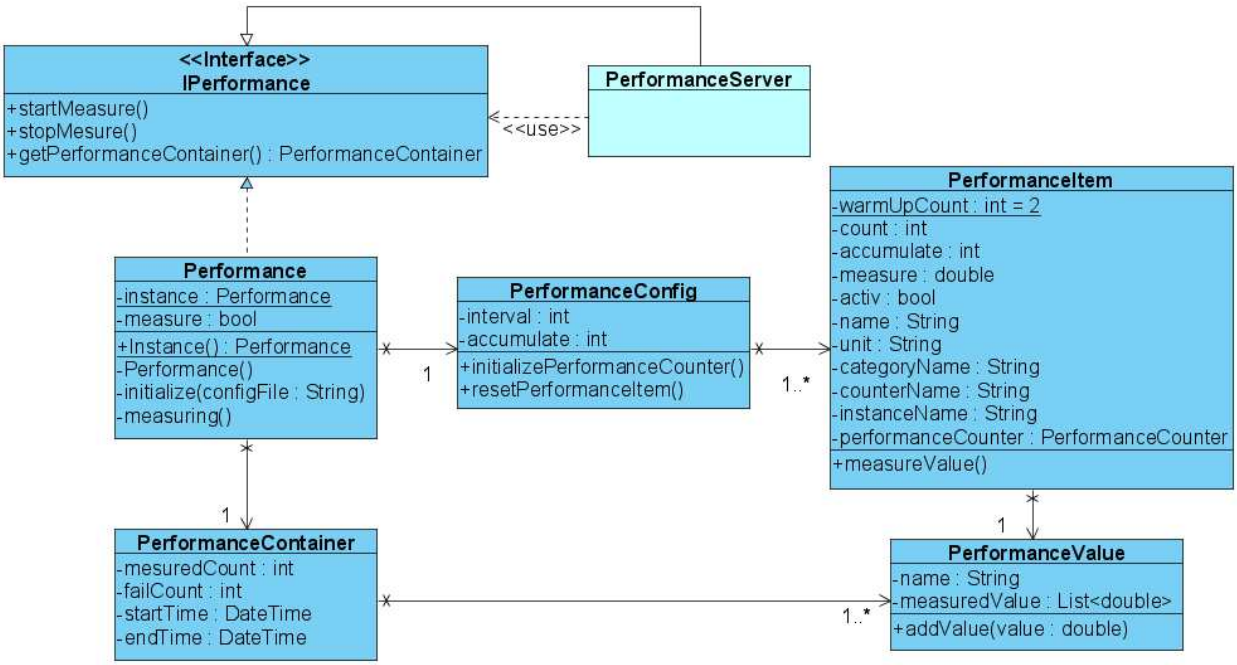


Abb. 3.4.5.1: UML-Klassendiagramm der Performance-Komponente.

Klasse Performance

Die Klasse *Performance* realisiert die Messung der Leistungsmetriken auf dem Flugdatenserver. Dabei reagiert Sie auf RPC-Aufrufe durch den *Controller*. Für die eingehende Kommunikation mit dem *Controller,* implementiert die Klasse *Performance* das Interface *IPerformance*.

Mit dem Aufruf der Methode *startMeasure()* durch den Controller in der Klasse *Performance* wird die Messung der Leistungsmetriken gestartet. Die Leistungsmetriken werden dabei in dem vordefinierten Intervall sequentiell gemessen und im dazugehörigen *PerformanceValue*-Objekt abgelegt. Mit dem Aufruf der Methode *stopMeasure()* wird die Messung der Leistungsmetriken wieder beendet. Durch den Aufruf der Methode *getPerformanceContainer(),* erfolgt die Übergabe der gemessenen Leistungsmetriken in Form eines *PerformanceContainer*-Objektes zurück an den *Controller*. Die Messung der Leistungsmetriken erfolgt einem separaten Thread über die Methode *measuring().*

Klassen PeformanceConfig / PerformanceItem

Die Klasse *PerformanceConfig* enthält die zu messenden Leistungsmetriken. Diese wird mit dem Start der *Performance-Anwendung* aus einer XML-Datei geladen, siehe Listing 3.4.3.2. Die Konfiguration umfasst das Intervall (in Sekunden) für die Messwertaufnahme, eine Akkumulierungskennzahl zur Zusammenfassung von Messwerten sowie sämtliche Leistungsmetriken, die während eines Analyselaufes gemessen werden sollen. Mit der Klasse *PerformanceItem* wird eine einzelne Leistungsmetrik repräsentiert. Für jede Leistungsmetrik, wird ein entsprechendes *PerformanceItem-Objekt* erstellt. Die zu messenden Leistungsmetriken können in der XML-Datei (siehe Listing 3.4.3.2) individuell für die Leistungsmessung aktiviert bzw. deaktiviert werden.

Mit der Methode *measureValue()* erfolgt für jedes *PerformanceItem* die Messwertaufnahme und Speicherung. Gespeichert werden die Messdaten in einem *PerformanceValue*-Objekt. Jeder *PerformanceItem* (Leistungsmetrik) hat dabei sein eigenes *PerformanceValue*-Objekt.

```
1  <?xml version="1.0"?>
2  <PerformanceConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4    <interval>1000</interval>
5    <accumulate>1</accumulate>
6    <performanceItemList>
7      <!-- System Metriken -->
8      <PerformanceItem>
9        <activ>true</activ>
10       <name>Thread Queue Länge</name>
11       <unit>count</unit>
12       <categoryName>System</categoryName>
13       <counterName>Processor Queue Length</counterName>
14       <instanceName></instanceName>
15     </PerformanceItem>
16     <PerformanceItem>
17       <activ>true</activ>
18       <name>Context Swichtes/sec</name>
19       <unit>count</unit>
20       <categoryName>System</categoryName>
21       <counterName>Context Switches/sec</counterName>
22       <instanceName></instanceName>
23     </PerformanceItem>
24     <PerformanceItem>
25       <activ>true</activ>
26       <name>P2FS.Server Aktive Threads</name>
27       <unit>count</unit>
28       <categoryName>Prozess</categoryName>
29       <counterName>Threadanzahl</counterName>
30       <instanceName>P2FS.Server</instanceName>
31     </PerformanceItem>
```

Listing 3.4.5.1: XML-Datei zur Konfiguration der zu messenden Leistungsmetriken.

Nachfolgend ein UML-Sequenzdiagramm, das den Ablauf der Methode *measuring()*, die in einem separaten Thread ausgeführt wird, darstellt. Diese Methode realisiert die Durchführung der Messung der verschiedenen Leistungsmetriken. Zur Vereinfachung der Darstellung, wird in dem folgenden Beispiel nur eine Leistungsmetrik (*PerformanceItem*) gemessen.

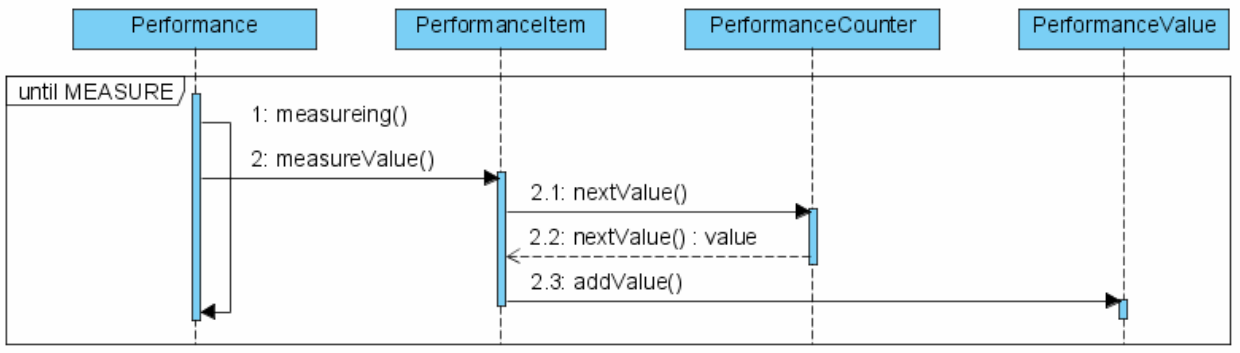


Abb. 3.4.5.2: UML-Sequenzdiagramm zur Messung der Leistungsmetriken.

Die Methode *measuring()* wird sequentiell in einem definierten Intervall solange ausgeführt, wie das Kennzeichen *MEASURE* zur Messung *TRUE* ist. In jedem Durchlauf erfolgt die Abfrage des aktuellen Wertes der entsprechenden Leistungsmetrik (*PerformanceItem*) über die Methode *measureValue()*. In dieser Methode wiederum erfolgt der Aufruf der Methode *nextValue() der Klasse PerformanceCounter*. Diese Methode liefert den aktuellen Wert zu einer bestimmten Leistungsmetrik zurück. Nach Erhalt des aktuellen Wertes, wird dieser im dazugehörigen *PerformanceValue*-Objekt abgelegt.

Klasse PerformanceServer

Bei der Klasse *PerformanceServer* handelt es sich um den *Server-Stub* der für die Kommunikation zwischen dem *Controller* und der Performance-Komponente benötigt wird. Im Vergleich zur *Worker-Komponente* ist hier eine unidirektionale Kommunikation ausreichend. Für die Realisierung des *Server-Stubs* implementiert und nutzt die Klasse *PerformanceServer* das Interface *IPerformance*.

3.4.6 ServerDummy-Komponente

Nachfolgend das vollständige UML-Klassendiagramm zur ServerDummy-Komponente:

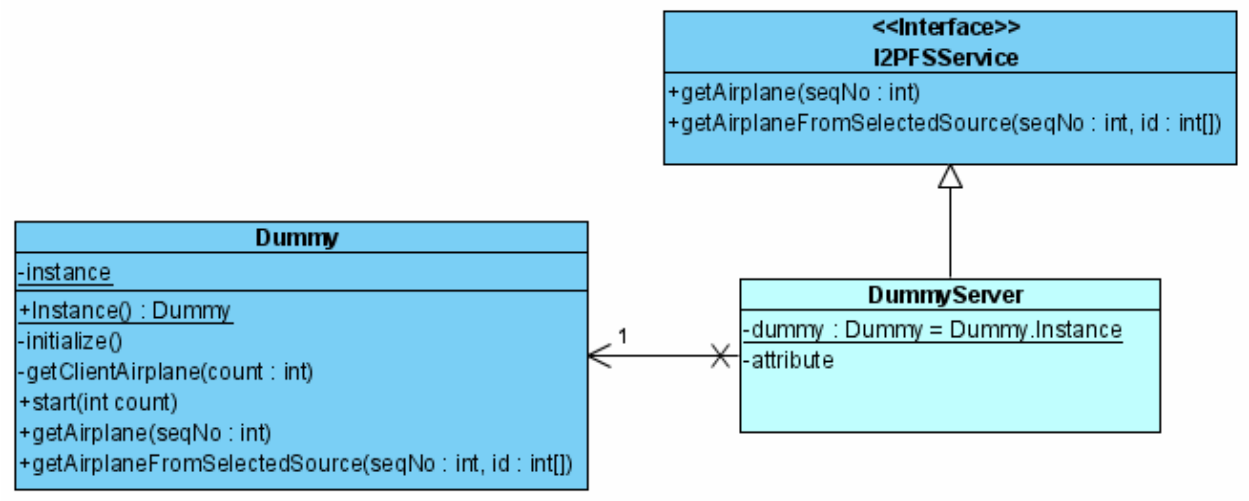


Abb. 3.4.6.1: UML-Klassendiagramm zur ServerDummy-Komponente.

Interface I2PFSService

Bei diesem Interface handelt es sich um die Schnittstellendefinition für die Kommunikation zwischen dem *ATC.Webserver* und dem *ATC.Server*. Der *ATC.Webserver* implementiert das Interface in seinem *Client-Stub,* der *ATC.Server* dagegen in seinem *Server-Stub*. Dieses Interface wird für die Realisierung des *DummyServers* benötigt. Um den *ATC.Server* zu ersetzen, wird der *DummyServer* von einem *Server-Stub genutzt. Dieser Server-Stub muss das Interface I2PFSService* implementieren.

Klasse Dummy

Die Klasse *Dummy* realisiert die Implementierung der beiden Methoden *getAirplane()* und *getAirplaneFromSelectedSoruce().* In der Dummy-Komponente ist die Funktionalität der beiden Methoden reduziert. Im Prinzip erfolgt bei jedem RPC-Aufruf einer Übergabe einer konstanten Menge an Flugzeugnachrichten (Airplane-Objekten). Die Anzahl die konstant übergeben wird, kann mit dem Programmstart über die Konsole fest vorgegeben werden.

Klasse DummyServer

Bei der Klasse *DummyServer* handelt es sich um den Server-Stub der ATC.Server Komponente des Flugdatenservers. Für die Realisierung implementiert die Klasse *DummyServer* das Interface *I2PFSService*. Zur Vereinfachung der Implementierung, nutzt die Klasse nicht wie üblich das Interface *IP2FSService*, sondern direkt die Klasse Dummy.

3.5 Implementierung

In diesem Kapitel werden die wichtigsten Bestandteile der Implementierung der verteilten Anwendung beschrieben. Grundsätzlich ist zu erwähnen, dass die gesamte vereilte Anwendung auf Basis des .NET-Frameworks und der Programmiersprache C# als dynamische Programm-Bibliothek (DLL) realisiert wurde. Für die Realisierung der verschiedenen Komponenten (*Controller, Worker, Performance und ServerDummy*) wurde diese Programm-Bibliothek verwendet.

3.5.1 Implementierung des .NET TcpChannel

Wie bereits ausführlich beschrieben, wird für die Kommunikation zwischen den Komponenten der verteilten Anwendung der *.NET RPC-Mechanismus TcpChannel* verwendet. Nachfolgend wird die Implementierung des *Server-Stubs* und des *Clients-Stubs* am Beispiel der Kommunikation zwischen der Controller- und der Performance-Komponente beschrieben. Es werden für jede unidirektionale Kommunikation zwischen zwei verteilten Komponenten die nachfolgenden Implementierungen benötigt.

- Gemeinsame Schnittstellendefinition (Interface)

Im nachfolgenden Listing 3.5.1.1 wird die Implementierung der gemeinsamen Schnittstellendefinition für zwei kommunizierende Komponenten dargestellt.

```
1  public interface IPerformance
2  {
3        void startMeasure();
4
5        void stopMeasure();
6
7        PerformanceContainer getPerformanceContainer();
8  }
```

Listing 3.5.1.1: Implementierung der gemeinsamen Schnittstellendefinition.

Um zu gewährleisten, dass die beiden Kommunikationspartner dieselben Methoden implementieren, die für die Remote-Aufrufe benötigt werden, ist das in Listing 3.5.1.1 dargestellte Interface nötig. Client-seitig implementiert der *Client-Stub* dieses Interface und server-seitig wird dasselbe Interface vom *Server-Stub* implementiert und gleichzeitig genutzt.

- Instanziierung des *TcpChannels* server-setitig

In Listing 3.5.1.2 wird die Instanziierung des *TcpChannels* dargestellt. Der nachfolgende Ausschnitt aus Listing 3.5.1.3 stellt dar, wie ein *TcpChannel* mit Hilfe der statischen Methode *Connect()* in der Klasse *Performance* instanziiert wird.

```
1  public static void Connect(int port, string applicationName, Type type,
2  string objectUri, string channelName)
3  {
4        IDictionary channelSettings = new Hashtable();
5        channelSettings["name"] = channelName;
6        channelSettings["port"] = port;
7
8
9        TcpChannel channel = new TcpChannel(channelSettings, null, null);
10       ChannelServices.RegisterChannel(channel, false);
11       RemotingConfiguration.CustomErrorsMode = CustomErrorsModes.Off;
12       RemotingConfiguration.ApplicationName = applicationName;
13       RemotingConfiguration.RegisterWellKnownServiceType(type,
14       objectUri, WellKnownObjectMode.SingleCall);
15 }
```

Listing 3.5.1.2: Instanziierung eines *TcpChannels*.

```
1  public class Performance : IPerformance
2  {
3        private void initialize()
4        {
5              try{
6                  // initialize TcpChannel
7                    ServiceProvider.Connect(9805, "Performance",
8                    typeof(PerformanceService), "service",
9                    "srvPerformance");
10             }
11             catch (Exception e) {}
12       }
```

Listing 3.5.1.3: Aufruf der Methode *Connect()* zur Initialisierung eines *TcpChannels*.

Die Instanziierung des Server-seitigen *TcpChannels* erfordert die Übergabe der nachfolgenden Parameter: Port-Nummer, Programm-Name (Alias), Typ des Remote-Objektes, Objekt-Url sowie einen Channel-Namen. Die Instanziierung erfolgt in der Klasse *Performance* der Komponente *Performance* und wird mit dem Start der Anwendung ausgeführt. Anhand der genannten Parameter, wird der *TcpChannel* in der Methode *Connect()* instanziiert und registriert. Die Angabe eines Channel-Namens ist nicht zwingend notwendig. Werden jedoch in einer Anwendung mehrere *TcpChannels* benötigt, müssen zur lokalen Unterscheidung der

verschiedenen *TcpChannel* zwingend Namen bei der Instanziierung angegeben werden. Wird kein Name angegeben, wird ein Standard-Name verwendet. Die Registrierung erfolgt über die Methode *RegisterChannel()* der Klasse *ChannelService*. Anhand der Port-Nummer, dem Programm-Namen (Alias) und der Objekt-Url, kann der *Client-Stub* das Remote-Objekt (*Server-Stub)* nutzen.

- Server-Stub (Server-seitig)

In Listing 3.5.1.3 wird die Implementierung des *Server-Stubs* dargestellt. Aus Platzgründen, wird nur eine Methode der drei implementierten Methoden dargestellt.

```
1  public class PerformanceService : MarshalByRefObject, IPerformance
2  {
3
4        private static readonly IPerformance performance =
5              Performance.Instance;
6
7        public void startMeasurement()
8        {
9              performance.startMeasurement();
10       }
11
```

Listing 3.5.1.3: Implementierung des *Server-Stub*.

Der *Server-Stub* muss die beiden Interfaces *MarshalByRefObjects* und *IPerformance* implementieren. Die Implementierung des Interface *MarshalByRefObjects* ermöglicht den Remote-Zugriff auf das Objekt bzw. auf die darin enthaltenen implementierten Methoden. Das Interface *IPerformance* stellt sicher, dass die Methoden zur Kommunikation im *Server-Stub* vorhanden sind. Die Referenz auf den Typ *IPerformance* wird genutzt, um die eingehenden Remote-Aufrufe an diese weiter zu delegieren. Die tatsächliche Implementierung der Methoden aus dem Interface *IPerformance* befinden sich dann in der Klasse *Performance*, nicht jedoch im *Server-Stub*, da dieser, wie in Abschnitt 3.3.3 beschrieben, nur als Kommunikationsdelegat agiert.

- Client-Stub (Client-seitig)

Im Listing 3.5.1.4 wird die Implementierung des *Clients-Stubs* dargestellt. Aus Platzgründen wird nur eine Methode der drei implementierten Methoden dargestellt.

```
1  public class PerformanceClient : IPerformance
2  {
3        private static PerformanceClient instance = new
4  PerformanceClient();
5        private TcpChannel channel;
6        private IPerformance performanceService;
7
8        private PerformanceClient(){
9              this.register();
10       }
11
12       public static PerformanceClient Instance{
13             get { return instance; }
14       }
15
16       private void register()
17       {
18             try {
19                 IDictionary channelSettings = new Hashtable();
20                 channelSettings["name"] = "clPerformance";
21
22                 channel = new TcpChannel(channelSettings, null, null);
23                 ChannelServices.RegisterChannel(channel, false);
24
25                 performanceService =
26                 (IPerformance)Activator.GetObject(typeof(IPerformance),
27                 "tcp://atcserver.et-it.fh-
28                 offenburg.de:9805/Performance/service");
29             }
30             catch (Exception e) {}
31         }
32
33         public void startMeasure()
34         {
35             try{
36                 performanceService.startMeasure();
37             }
38             catch (Exception e) {}
```

Listing 3.5.1.4: Implementierung des *Client-Stub.*

Der *Client-Stub* ist als Singleton-Klasse implementiert, was zur Folge hat, dass nur eine Instanz der Klasse zu jedem Zeitpunkt existiert. Der Zugriff auf diese Instanz erfolgt über das statische Property *Instance.* Mit dem Instanziieren des *Client-Stubs* erfolgt der Aufruf der Methode *register()*. In dieser Methode wird einmalig ebenfalls ein lokaler *TcpChannel* instanziiert und registriert. Über diesen *TcpChannel* erfolgt später die Kommunikation mit dem *Server-Stub*. Um Zugriff auf das Remote-Objekt zu erhalten, erfolgt zudem der Aufruf der Methode *GetObject()*

der Klasse *Activator*. Für die Instanziierung muss der Typ und die Url des Remote-Objektes übergeben werden. Die Url setzt sich zusammen aus der IP-Adresse bzw. DNS-Namen, der Port-Nummer, dem Anwendungs-Namen und der Objekt-Url. Damit wird der Remote-Objekt-Zugriff ermöglicht.

3.5.2 Implementierung zur Messung der Leistungsmetriken

Die Messung der Leistungsmetriken wird realisiert durch die im .NET-Framework bereitgestellte Klasse *PerformanceCounter* aus der Namespace *System.Diagnostics*. Wie bereits in Abschnitt 2.3.3 beschrieben, können mit dieser Klasse sämtliche auf einem System zur Verfügung stehenden Leistungsmetriken abgerufen werden. Wie bereits in Abschnitt 3.4.5 dargestellt, wird für jede zu messende Leistungsmetrik ein solches *PerformanceCounter*-Objekt verwendet.

In dem nachfolgenden Listing 3.5.2.1 wird die Instanziierung der *PerformanceCounter*-Objekte dargestellt.

```
1  public void initializePerformanceCounter()
2  {
3        List<PerformanceItem> activItemList = new List<PerformanceItem>();
4
5        foreach (PerformanceItem item in performanceItemList)
6        {
7              if (item.Activ)
8              {
9                    item.Accumulate = accumulate;
10                     item.PerformanceValue = new
11 PerformanceValue(item.Name);
12                     item.PerformanceCounter = new
13 PerformanceCounter(item.CategoryName, item.CounterName,
14 item.InstanceName);
15                     activItemList.Add(item);
16                   }
17       }
18
19       performanceItemList = activItemList;
```

Listing 3.5.2.1: Implementierung zur Instanziierung der *PerformanceCounter*.

Wie in Kapitel 3.4.5 dargestellt, werden die zu messenden Leistungsmetriken über eine XML-Datei konfiguriert und in der *Performance*-Komponente eingelesen. Nach dem Einlesen werden die Leistungsmetriken als *PerformanceItem*-Objekt abgebildet. Für jedes dieser

PerformanceItem-Objekte muss das entsprechende *PerformanceCounter*-Objekt instanziiert werden. Dies erfolgt in der Methode *initializePerformanceCounter()* die unmittelbar nach dem Einlesen der XML-Datei ausgeführt wird. Für jedes *PerformanceItem*-Objekt wird ein *PerformanceCounter*-Objekt instanziiert. Für die Instanziierung müssen lediglich die Parameter *CategoryName*, *CounterName* und *InstanceName* übergeben werden.

Im nachfolgenden Listing 3.5.2.2 wird die eigentliche Messung der Leistungsmetriken bzw. einer Leistungsmetrik dargestellt.

```
1  public void measureValue()
2  {
3        double value = Math.Round(performanceCounter.NextValue(), 2);
4
5        count++;
6
7        // warm-up
8        if(count > warmUpCount)
9        {
10             measure += value;
11
12             if (((count-warmUpCount)%accumulate) == 0)
13             {
14                   performanceValue.addValue(measure / accumulate);
15                   measure = 0;
16             }
17       }
18 }
```

Listing 3.5.2.2: Implementierung der Abfrage der *PerformanceCounter*.

Die Messung des aktuellen Wertes einer Leistungsmetrik erfolgt über die Methode *measureValue()* aus der Klasse *PerformanceItem*. In dieser Methode erfolgt wiederum der Aufruf der Methode *nextValue()* aus der Klasse *PerformanceCounter*. Diese Methode nimmt eine aktuelle Messung vor und liefert den Messwert zurück. Befindet sich die Leistungsmessung noch in der sogenannten *Warm-Up-Phase*, erfolgt keine Messwert-aufnahme. Die *Warmp-Up-Phase* kann beliebig definiert werden, umfasst aber aktuell die ersten beiden Messungen. Nach der *Warm-Up-Phase* werden die Messwerte akkumuliert und entsprechend im dazugehörigen *PeformanceValue*-Objekt abgelegt.

3.5.3 Implementierung zur Messung der Antwortzeiten

Im nachfolgenden Listing 3.5.3.1 wird ein Ausschnitt zur Messung der Antwortzeiten auf die Client-Anfragen dargestellt.

```
1  private void receiving()
2  {
3        Thread.Sleep(this.waitTime);
4
5        while(RUNNING)
6        {
7              workerContainer.incClientFlag(ClientFlags.reqCount);
8
9
10             if (Worker.TRACE)
11                   TraceLog.logTrace("Client " + id, "receiving", "mp1 -
12                   request" + " seqNo: " + nextSeqNo);
13
14
15             stopWatch.Reset();
16             stopWatch.Start();
17
18             ClientAirplane clientAirplane = (ClientAirplane)
19             clientService.getAirplane(nextSeqNo);
20
21             stopWatch.Stop();
22
23
24             double elapsedTime = stopWatch.Elapsed.TotalMilliseconds;
25             int countAirplane = clientAirplane.AirplaneList.Count;
```

Listing 3.5.3.1: Implementierung zur Messung der Antwortzeiten.

Für die Messung der Antwortzeiten auf die Client-Anfragen wird die im .NET-Framework verfügbare Klasse *StopWatch* aus dem *Namespace System.Diagnostics* verwendet. Die Klasse *StopWatch* ermöglicht die Messung der in Kapitel 2.3.2 beschriebenen *ElapsedTime* (Laufzeit + Wartezeit) einer Funktion. Die Klasse *StopWatch* unterstützt dabei einen High-Resoultion-Timer, womit extrem genaue Zeitmessungen möglich sind.

Zur Messwertaufnahme wird die Implementierung instrumentiert. Vor dem Aufruf der zu messenden Methode erfolgen die Methodenaufruf *Reset()* und *Start()* aus der Klasse *StopWatch*. Damit wird das *StopWatch*-Objekt zurückgesetzt und die Zeitmessung gestartet. Anschließend erfolgt der eigentliche Methodeaufruf der zu messenden Methode *getAirplane()*. Direkt nach der Terminierung dieser Methode wird die Zeitmessung mit der Methode *Stopp()* wieder gestoppt. Über das Property *Eleapsed.TotalMilliseconds* kann der gemessene Zeitwert in Millisekunden dann abgefragt werden.

3.6 Ergebnis

In diesem Kapitel wird das Ergebnis der entwickelten Anwendung zur Leistungsanalyse dargestellt. Zudem werden die Messdaten, die mit dieser verteilten Anwendung gemessen werden können beschrieben.

3.6.1 Verteilte Anwendung

Jede Komponente der verteilten Anwendung wurde als separater Prozess implementiert. Der *Controller* wiederum wurde für die Analyse des *Gesamtsystems* und des *Teilsystems* aufgeteilt in zwei getrennte Prozesse. Alle Komponenten wurden als Konsolenanwendung realisiert.

In der nachfolgenden Abbildung 3.6.1.1 wird die Konsolenanwendung zur Analyse des *Gesamtsystems* dargestellt. Die beiden Analyseanwendungen (*AnalysisOverall* und *AnalysisSub*) unterscheiden sich lediglich in der Instanziierung der entsprechenden Klasse (*AnalysisOverallSystem* oder *AnalysisSubsystem*).

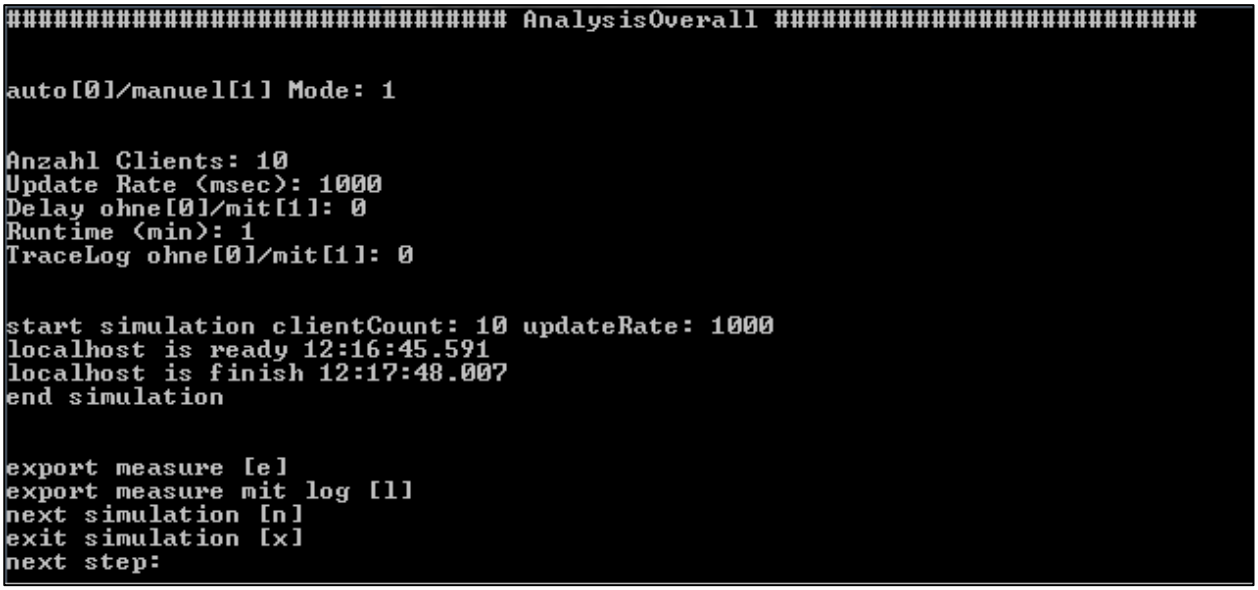


Abb. 3.6.1.1: Konsolenanwendung zur Analyse des Gesamtsystems.

Mit dem Start der Anwendung erfolgt die Auswahl zur manuellen oder automatischen Durchführung. Im Fall der automatischen Durchführung der Analyse, wird die entsprechende XML-Datei mit den Konfigurationen für die Analyseläufe eingelesen. Danach werden die eingelesenen Analyseläufe sequentiell nacheinander ausgeführt.

Bei der manuellen Ausführung kann durch die Eingabe einer Konfiguration manuell ein Analyselauf initiiert werden. Hierfür müssen die in der Abbildung 3.6.1.1 dargestellten Parameter (*Anzahl Clients, Update Rate, Delay-Einstellung, Runtime* und *TraceLog*) eingegeben werden. Grundsätzlich können beliebig viele Analyseläufe nacheinander durchgeführt werden. Nach jedem Analyselauf ist es möglich die Messdaten (Messwerte der Performance-Komponente und der *Worker-Komponenten*) im CSV-Format zu exportieren. Im nachfolgenden Abschnitt 3.6.2 werden die generierten Messdaten ausführlich beschrieben.

In der Konsolenanwendung der *Worker-Komponente* und der *Performance-Komponente* findet keine Benutzerinteraktion statt. Die beiden Prozesse dienen nur dazu, auf die RPC-Aufrufe der Analyseanwendung (*AnalysisOverall* und *AnalysisSub*) zu reagieren. In den beiden nachfolgenden Abbildungen 3.6.1.2 und 3.6.1.3 werden die beiden Konsolenanwendungen dargestellt.

Abb. 3.6.1.2: Konsolenanwendung der Worker-Komponente.

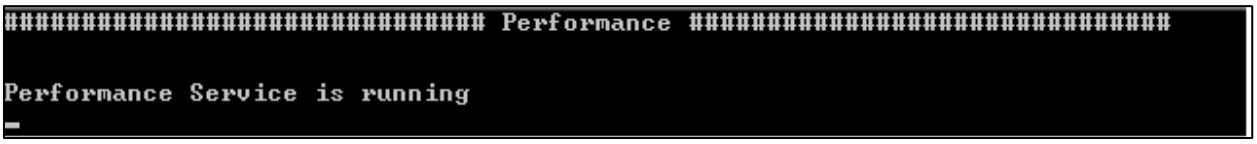


Abb. 3.6.1.3: Konsolenanwendung der Peformance-Komponente.

In der Konsolenanwendung der *ServerDummy-Komponente* erfolgt die Eingabe der Anzahl der Flugzeugnachrichten, die bei den eingehenden Anfragen des *ATC.Webservers* statisch für den Hüllentest übergeben werden sollen. Damit lässt sich unter anderem Testen, wie sich der Flugdatenserver bei einer größeren Datenmenge (Flugzeugnachrichten) verhält. In der Abbildung 3.6.1.4 wird die Konsolenanwendung des *ServerDummys* dargestellt.

Abb. 3.6.1.4: Konsolenanwendung der ServerDummy-Komponente.

3.6.2 Messdaten

In diesem Kapitel werden die generierten Messdaten aus der verteilten Anwendung genauer beschrieben.

Grundsätzlich werden für jeden ausgeführten Analyselauf Messdaten erhoben und dem Controller übergeben. Der Controller wiederum nimmt eine Zuordnung der Messdaten zum entsprechenden Analyselauf bzw. zu den Parametern der bei einem Analyselauf verwendeten Konfiguration. Die Messdaten können als CSV-Format nach jedem Analyselauf exportiert werden. Mit jedem Export werden alle gesammelten Messdaten der durchgeführten Analyseläufe in eine CSV-Datei geschrieben, die somit beliebig viele Messdaten von diesen Analyseläufen enthalten kann.

In der nachfolgenden Abbildung 3.6.2.1 wird eine in Microsoft Excel importierte CSV-Datei mit Messdaten angezeigt. Die rot markierten Bereiche markieren die vier Bestandteile der Messdaten eines Analyselaufes.

	A	B	C	D	E	F	G
1	#Id: 1	StartTime:	10:57:56.949	EndTime:	11:01:00.40	RequestCour	2680
2	#Konfiguration	ClientCount:	50	Update Rate:	1000	Delay:	1
3	## Skalierung	1<50	50<100	100<150	150<200	200<300	300<400
4	## ResponseTime	8918	12	2	3	2	0
5	## ResponseAirplane	58	8879	0	0	0	0
6	### CPU Auslastung %	19,23	20	20,77	21,54	19,23	16,92
7	### P2FS.Server CPU Time %	9,23	16,92	12,31	12,31	12,31	16,92
8	### w3wp iis cpu time %	18,46	18,46	23,08	23,08	10,77	10,77
9	### RAM Verfügbare MB	1128	1128	1129	1129	1128	1128
10	#### Log	startTime	endTime	Measure/Request Cou	clientCount	REQ_COUN	REQ_FAIL_COU
11	#### Performance	10:57:58.261	11:00:59.121	178			0
12	#### pc-1c	10:57:58.230	11:00:59.121	893	5	900	0
13	#### pc-1d	10:57:58.230	11:00:59.136	894	5	899	0
14	#### pc-2c	10:57:58.230	11:00:59.136	893	5	900	0
15	#End						
16	#Id: 2	StartTime:	11:01:10.496	EndTime:	11:04:13.58	RequestCour	5323
17	#Konfiguration	ClientCount:	100	Update Rate:	1000	Delay:	1
18	## Skalierung	1<50	50<100	100<150	150<200	200<300	300<400
19	## ResponseTime	17455	104	53	45	32	16
20	## ResponseAirplane	3	17648	90	0	0	0
21	### CPU Auslastung %	30	39,23	38,46	41,54	40,77	42,31
22	### P2FS.Server CPU Time %	24,62	20	24,62	32,31	32,31	24,62
23	### w3wp iis cpu time %	23,08	41,54	35,38	30,77	35,38	49,23
24	### RAM Verfügbare MB	1128	1128	1128	1128	1128	1128
25	#### Log	startTime	endTime	Measure/Request Cou	clientCount	REQ_COUN	REQ_FAIL_COU
26	#### Performance	11:01:11.777	11:04:12.183	177			0
27	#### pc-1c	11:01:11.761	11:04:12.199	1775	10	1786	0
28	#### pc-1d	11:01:11.761	11:04:12.214	1773	10	1785	0
29	#### pc-2c	11:01:11.761	11:04:12.230	1775	10	1785	0
30	#End						

Abb. 3.6.2.1: Exportierte Messdaten im CSV-Format.

1. Messdaten-Header

Der Messdaten-Header leitet die Messdaten eines Analyselaufes ein. Er enthält eine laufende Nummer, den Start- und End-Zeitpunkt, sowie die Konfiguration (Anzahl simulierter Benutzer, Update-Rate, Delay-Einstellung) des Analyselaufes.

2. Messdaten der Worker-Komponente

Nach dem Messdaten-Header folgen die Messdaten der *Worker*. Die Messdaten umfassen die Antwortzeiten (*ResponseTimes*) und die erhaltenen Flugzeugnachrichten (*ResponseAirplanes*) auf die Client-Anfragen. Die Messdaten der einzelnen *Worker* werden wie in Abschnitt 3.4.2 kategorisiert gespeichert und im *Controller* zusammengeführt.

3. Messdaten der Performance-Komponente

Die Messdaten der *Performance*-Komponente umfassen eine Messreihe für jede Leistungsmetrik. Die Messwerte werden nicht kategorisiert und somit direkt abgespeichert. Im Beispiel aus Abbildung 3.6.2.1 wurden Messwerte von vier Leistungsmetriken aufgenommen.

4. Auswertung zum Analyselauf

Für jeden Analyselauf werden zu Kontrollzwecken wichtige Informationen festgehalten, wie Start- und Endzeitpunkt der Leistungsmessung, die Anzahl der ausgeführten Messwert-aufnahmen bzw. Client-Anfragen sowie die Anzahl der fehlerhaften Messwertaufnahmen bzw. Client-Anfragen. Anhand dieser Informationen können im Fehlerfall wichtige Rückschlüsse gezogen werden. Damit ist das Verhalten eines Analyselaufes jederzeit transparent.

4 Leistungsanalyse

In diesem Kapitel wird die Durchführung der Leistungsanalyse beschrieben. Im ersten Abschnitt werden die Anforderungen an die Leistungsanalyse genannt, danach wird die Testumgebung, in der die Leistungsanalyse durchgeführt wird, der Analysevorgang selber und dessen Ergebnisse beschrieben.

4.1 Anforderungen

Nachfolgend werden die Anforderungen zur Leistungsanalyse genannt. Teilweise überschneiden sich diese mit den Anforderungen aus Abschnitt 3.1 (Entwicklung der Anwendung zur Leistungsanalyse).

- Die Leistungsanalyse soll mit einer *Simulation der Benutzerlast* und einer Messung von Leistungsmetriken erfolgen.

- Die *Simulation der Benutzerlast* soll mit verschiedenen Konfigurationen (Anzahl zu simulierende Benutzer, Delay-Einstellung, Update-Rate) ausgeführt werden.

- Die Leistungsmessung soll während einer *Simulation der Benutzerlast* erfolgen, um das Verhalten des Flugdatenservers bei einer bestimmten bzw. steigenden Benutzerlast zu ermitteln. Damit soll die maximale Benutzerlast (*Peak Load*) ermittelt werden.

- Die Leistungsmessung auf dem Flugdatenserver soll mindestens die Messwertaufnahme von Leistungsmetriken wie CPU-, Arbeitsspeicher-, Netzwerkschnittstellen-Auslastung und die Antwortzeiten der Client-Anfragen auf dem *Worker* beinhalten.

- Mit der Leistungsanalyse sollen mögliche Leistungsengpässe identifiziert werden.

4.2 Testumgebung

Wie in Abschnitt 3.2.1 beschrieben, sind für die *Simulation der Benutzerlast* mehrere Rechner (Clients) vorgesehen. Jeder dieser Rechner simuliert eine ihm zugewiesene Menge an Benutzern.

Als Testumgebung für die *Simulation der Benutzerlast* stand das Labor der Fakultät E+I bzw. der Angewandten Informatik im Steinbeis Gebäude zur Verfügung. Das Labor verfügt über insgesamt 23 vernetzte Rechner, die für die *Simulation der Benutzerlast* verwendet werden können. In Abbildung 4.2.1 wird die Netzwerktopologie dieses Labors dargestellt. Technische Details können aus der Abbildung nicht entnommen werden, vielmehr geht es darum, einen groben Überblick über die Topologie des Netzwerkes zu erhalten.

Die einzelnen Rechner verfügen über jeweils zwei Netzwerkschnittstellen. Die erste Netzwerkschnittstelle dient zur Verbindung zum öffentlichen Netzwerk (PUBLICNET) der Hochschule. Mit der zweiten Netzwerkschnittstelle werden kleine lokale Netze (Inseln), wie in Abbildung 4.2.1 dargestellt, gebildet. Für die Durchführung der Simulation, wurde das öffentliche Netzwerk der Hochschule verwendet, da über dieses der Flugdatenserver erreichbar ist. Das lokale Netzwerk wurde für die Leistungsanalyse nicht benötigt. Für die Ausführung von Leistungstests sollte grundsätzlich geprüft werden, inwiefern das zu verwendende Netzwerk selber zu einem Leistungsengpass werden kann. Dies ist beim Hochschul-Netzwerk nicht der Fall, da der erwartete Datendurchsatz bei der Ausführung der Simulation weitaus geringer ist als die verfügbare Bandbreite von 1 Gbit. Durch die in Abschnitt 4.4 dargestellten Ergebnisse, wird diese Einschätzung bestätigt.

Die einzelnen Rechner selbst sind jeweils mit einer 100-Mbit Nerzwerkkarte ausgestattet, vereinzelt sogar mit einer 1 Gbit Netzwerkkarte. Die Rechner sind an das Hochschul-Netzwerk über vier 100 Mbit Switches verbunden, die wiederum über die 1 Gbit Verbindung mit dem 1 Gbit Switch des Rechenzentrums verbunden sind. Mit diesem 1 Gbit Switch ist auch der Flugdatenserver verbunden. Weiterhin sind die Rechner mit ausreichend Systemressourcen (CPU und Arbeitsspeicher) ausgestattet, so dass hier keinerlei Leistungsengpässe zu erwarten sind. Es hat sich herausgestellt, dass die Ausführung der *Worker* auf den einzelnen Rechnern nur 5-10% CPU-Auslastung verursacht. Somit sind die vorhandenen Ressourcen in der Testumgebung für die Durchführung der Leistungsanalyse bzw. *Simulation der Benutzerlast* ausreichend.

In der nachfolgenden Abbildung 4.2.1, wird die Netzwerktopologie des Angewandte Informatik Labors im Steinbeis Gebäude dargestellt.

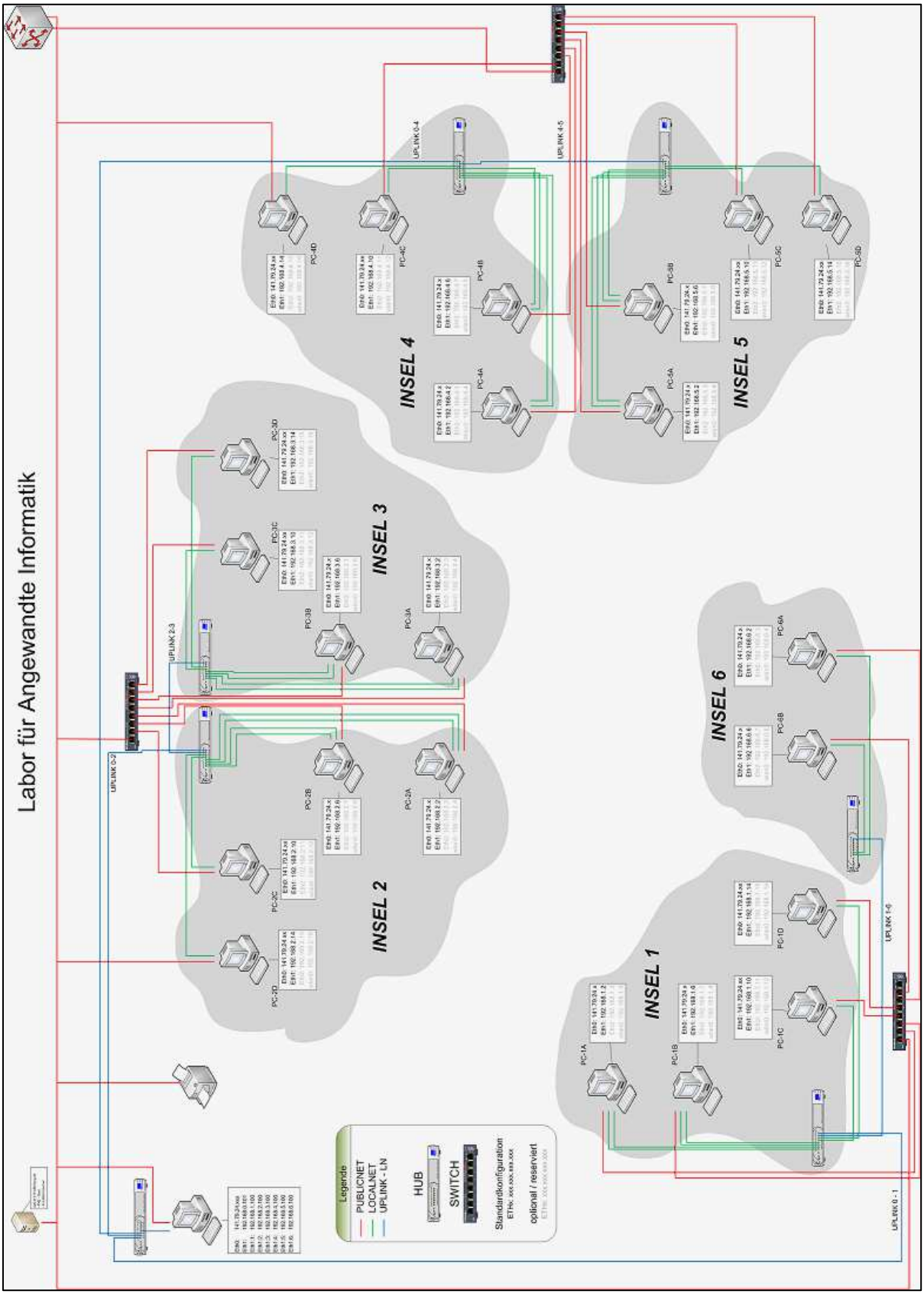


Abb. 4.2.1: Netzwerktopologie im Labor Angewandte Informatik.

4.3 Vorgehensweise zur Leistungsanalyse

In diesem Kapitel wird die Vorgehensweise zur Leistungsanalyse beschrieben. Die Leistungsanalyse erfolgt mit der Ausführung der *Simulation der Benutzerlast* und der Messwertaufnahme verschiedener relevanter Leistungsmetriken. Die *Simulation der Benutzerlast* soll, wie in den Anforderungen in Abschnitt 4.1 beschrieben, mit verschiedenen Konfigurationen ausgeführt werden. Das Ziel ist es zu verschiedenen Testszenarien bzw. Testfällen Messreihen zu generieren, die zur Analyse des Leistungsverhaltens des Flugdatenservers verwendet werden können. Für die Durchführung der *Simulation* wird der in Abschnitt 2.3.4 beschriebene *Capacity Test* verwendet. Hierbei wird ermittelt, wie sich der Flugdatenserver mit einer steigenden Zahl an Benutzern verhält. Zudem kann damit der *Peak Load*, also die maximale Benutzerlast, ermittelt werden. Mit diesen Erkenntnissen können Leistungsengpässe identifiziert und Aussagen getroffen werden welcher Menge an Benutzern der Flugdatenserver später zur Verfügung gestellt werden kann.

Zur Vorbereitung der Leistungsanalyse wurde der in Abschnitt 2.3.4 beschriebene *Performance Testing* Prozess zur Orientierung verwendet. Für eine korrekte Analyse, müssen erst die relevanten Testszenarien ermittelt werden. Die zu analysierende Hauptfunktionalität der Web-Anwendung ist die ständige Bereitstellung der Flugzeugnachrichten. Diese Datenbereitstellung an die Web-Clients wird unter einer bestimmten Benutzerlast getestet. Dabei handelt es sich um die Basisfunktionalität der Flugdatenserver Web-Anwendung. Weitere Funktionalitäten innerhalb des Web-Clients, z.B. das Filtern von Flugzeugnachrichten, werden dabei nicht weiter betrachtet, da sie für die Leistungsanalyse unerheblich sind. Vielmehr geht es darum eine wachsende Benutzerlast zu simulieren. Dies ist mit dem Test der Basisfunktionalität gewährleistet.

Als Ausgangsbasis bzw. Testszenarien für die Definition von Testfällen können die beiden Analyse-Modi *Gesamtsystem* (mit und ohne Hüllentest) und *Teilsystem* verwendet werden. So ergeben sich die drei nachfolgenden Testszenarien:

- Gesamtsystem
- Gesamtsystem als Hüllentest (*DummyServer*)
- Teilsystem

Auf Basis dieser Testszenarien wurden jeweils verschiedene Testfälle abgeleitet. Die Testfälle wiederum können aus den verschiedenen Konfigurationen, mit denen die *Simulation der Benutzerlast* ausgeführt wird, ermittelt werden. So ergibt sich für jede relevante Kombination der Konfigurationsparameter ein Testfall. Die Konfiguration einer Simulation beinhaltet die nachfolgenden Parameter:

- Anzahl zu simulierende Benutzer
- Update-Rate (Intervall der Client-Anfragen)
- Delay-Einstellung (mit oder ohne)

Im Vorfeld der Definition der Testfälle wurden bereits stichprobenartig erste Tests zur Simulation ausgeführt, um ungefähr das Verhalten des Flugdatenservers einschätzen und damit möglichst sinnvolle Testfälle definieren zu können. Erwartungsgemäß wurde dabei festgestellt, dass sich die beiden Testszenarien *Gesamtsystem* und *Teilsystem* im Verhalten bezüglich der Benuterzlast stark unterscheiden, so dass hier der Test von unterschiedlich hohen Benutzerlasten sinnvoll ist. Dieser Unterschied resultiert daher, das bei der Analyse des *Gesamtsystems* die CPU von beiden Flugdatenserver-Komponenten (*ATC.Server* und *ATC.Webserver*) beansprucht wird, bei der Analyse des *Teilsystems* dagegen nur die *ATC.Server-Komponente.* In der nachfolgenden Tabelle 4.3.1 werden die verschiedenen Konfigurationsvarianten für die verschiedenen Testszenarien dargestellt.

Delay-Einstellung		**Update-Rate in Millisekunden**		**Anzahl zu simulierender Benutzer**	
Gesamt / Teil	**Gesamt Hülle**	**Gesamt / Teil**	**Gesamt Hülle**	**Gesamt / Gesamt Hülle**	**Teil**
mit Delay	mit Delay	1000	1000	50	150
ohne Delay		5000		100	225
		10000		150	300
		15000		200	375
				250	450
				300	525
				350	600
				400	675
				450	750
				500	825
				550	900
				600	975
				650	1050
				700	1125

Tabelle 4.3.1: Konfigurationsvarianten für die verschiedenen Testszenarien.

Da die Konfigurationsvarianten für die verschiedenen Testszenarien zum Teil gleich sind, werden diese in der Tabelle 4.3.1 gemeinsam dargestellt.

Die zwei Varianten der Delay-Einstellung (mit oder ohne Delay) werden in den Testszenarien *Gesamtsystem* und *Teilsystem* berücksichtigt. Im Testszenario *Gesamtsystem als Hüllentest* wird nur eine Variante verwendet, da keinerlei Unterschiede bei der Datenbereitstellung durch den *DummyServer* in den beiden Delay-Einstellungen bestehen.

Bei der Update-Rate werden im Testszenario *Gesamtsystem* und *Teilsystem* die vier wichtigsten Varianten getestet. Update-Raten, die darüber liegen, sind aufgrund der zu langen Aktualisierungszeit für die Darstellung der Flugbewegungen ungünstig. Durch den Test dieser vier Varianten kann der Einfluss der Update-Rate auf das Verhalten des Flugdatenserver ermittelt werden. Für das Testszenario *Gesamtsystem als Hüllentest* wird nur eine Update-Rate von 1000 ms getestet. Der Einfluss der Update-Rate sollte über die Testszenarien *Gesamtsystem* und *Teilsystem* bereits ermittelt werden, so dass dies im Hüllentest keine Rolle spielt.

Die Testszenarien *Gesamtsystem* und *Gesamtsystem als Hüllentest* verwenden jeweils die gleiche Anzahl an zu simulierenden Benutzern. Im Testszenario *Teilsystem* dagegen wird eine höhere Abstufung verwendet, um eine höhere Anzahl an Benutzern mit weniger Testfällen abzudecken.

Die Testfälle für eine Testszenario können anhand der Tabelle 4.3.1 durch Verwendung des Kreuzproduktes mit den verschienen Varianten der Konfiguration abgeleitet werden. Die Testfälle werden alle automatische durch die bereits beschriebene *automatische Analyse* durchgeführt. Die Laufzeit jedes Analyselaufes bzw. Testfall beträgt dabei 3 Minuten.

Ein wichtiger Bestandteil des *Performance Testing* Prozess besteht darin, die Leistungsmetriken zu bestimmen die während eines Testlaufes gemessen werden. Für die gerade genanten Testfälle werden alle die im Kapitel 2.3.2 beschrieben Leistungsmetriken gemessen. Aus diesem Grund werden die Leistungsmetriken in diesem Abschnitt nicht mehr beschrieben.

Ein *Profiling-Tool* wird für die Leistungsanalyse nicht eingesetzt, da sich kein geeignetes und günstiges Tool hat finden lassen. Frei verfügbare Tools, die getestet wurden, haben sich leider als nicht sehr hilfreich herausgestellt.

4.4 Ergebnisse und Analyse

4.4.1 Einführung

In diesem Abschnitt werden die Ergebnisse der Leistungsanalyse beschrieben und analysiert. Hierfür wird das Verhalten von verschiedenen Leistungsmetriken dargestellt und bewertet. Da für die Leistungsanalyse zusätzlich kein *Profiling Tool* eingesetzt wurde, können teilweise die Ursachen für ein bestimmtes Verhalten nicht exakt geklärt werden. In den nachfolgenden Abschnitten werden die Ergebnisse der drei Testszenarien jeweils separat betrachtet. Anschließend folgt ein Fazit zu den Ergebnissen der Leistungsanalyse. Die Leistungsanalyse für die drei Testszenarien wurde auf Basis der in Abschnitt 4.3 definierten Testfälle (siehe Tabelle 4.3.1) durchgeführt.

Wie bereits in Abschnitt 4.3 erwähnt, wurden für die Leistungsanalyse eine Vielzahl von Leistungsmetriken gemessen. Bei der Darstellung der Ergebnisse werden jedoch nur die relevanten Messergebnisse der nachfolgenden Leistungsmetriken aus Tabelle 4.4.1.1 bewertet.

Leistungsmetrik	Beschreibung
ASP.NET Anfragen pro Sekunde	Anzahl der verarbeiteten Client-Anfragen pro Sekunde auf dem *ATC.Webserver*.
.NET Remote Anfragen pro Sekunde	Anzahl der Anfragen vom *ATC.Webserver* zum *ATC.Server* pro Sekunde.
Kontextwechsel pro Sekunde	Anzahl der Kontextwechsel, die pro Sekunde auf dem Flugdatenserver ausgeführt werden.
Anzahl Flugzeugnachrichten pro Anfrage	Anzahl Flugzeugnachrichten, die ein Benutzer auf eine Client-Anfrage erhält.
Gesendete Bytes pro Sekunde	Anzahl der gesendete Bytes, die der Flugdatenserver über seine Netzwerkschnittstelle pro Sekunde versendet.
ASP.NET Ausführungszeiten	Verarbeitungszeiten der Client-Anfragen auf dem *ATC.Webserver.*
Antwortzeiten	Antwortzeiten auf die Client-Anfragen der simulierten Benutzer.
ATC.Server CPU-Zeit	Anteil verwendeter CPU-Zeit des *ATC.Servers.*
ATC.Webserver CPU-Zeit	Anteil verwendeter CPU-Zeit des *ATC.Webservers.*
CPU-Auslastung	CPU-Auslastung auf dem Flugdatenserver
Verwendeter Arbeitsspeicher	Verwendeter Arbeitsspeicher auf dem Flugdatenserver

Tabelle. 4.4.1.1 Leistungsmetriken, deren Messwerte in der Leistungsanalyse erfasst werden.

Vor der Beschreibung der Ergebnisse kann bereits vorweg genommen werden, dass sich erhebliche Unterschiede in der Leistung des Flugdatenserver zwischen den beiden Delay-Einstellungen (mit oder ohne) ergeben haben. So können *ohne Delay* vom Flugdatenserver größere Benutzerlasten verarbeitet werden, als mit Delay. Mit der Delay-Einstellung wird wie bereits beschrieben ein bestimmter Web-Client simuliert. Mit der Einstellung *mit Delay* wird ein *Standard-Web-Client* und *ohne Delay* eine *Privilegierter-Web-Client* simuliert. Wie bereits in Kapitel 2.2.4 ausführlich beschrieben, bestehen in der Bereitstellung der Flugzeugnachrichten zwischen den beiden Delay-Einstellungen nachfolgende Unterschiede:

- Bereitstellung der Flugzeugnachrichten auf dem *ATC.Server*
  Für einen *Standard-Web-Client* (*mit Delay*) muss bei jedem Aufruf die Menge der Flugzeugnachrichten berechnet werden. Für den *Privilegierten-Web-Client* (*ohne Delay*) ist das nicht notwendig. Die Bereitstellung von Flugzeugnachrichten im *ATC.Server ohne Delay* ist damit weniger aufwändig, als mit Delay.

- Optimierungsmaßnahme auf dem *ATC.Webserver*
  Im *Privilegierten-Web-Service* (*ohne Delay*) ist eine Optimierungsmaßnahme implementiert, die die Menge an Flugzeugnachrichten, die an den Web-Cilent zurückgegeben wird, reduziert. Im *Standard-Web-Service* (*ohne Delay*) ist diese nicht vorhanden, so dass alle Flugzeugnachrichten an den Web-Client zurückgegeben werden.

Aufgrund der gravierenden Unterschiede in der Leistung des Flugdatenservers zwischen den beiden Delay-Einstellungen, wird bei der Darstellung der Ergebnisse in den Testszenarien *Gesamtsystem* und *Teilsystem* explizit darauf eingegangen. Damit kann nachvollzogen werden, inwiefern sich die genannten Unterschiede der Datenbereitstellung auf die Leistung auswirken. Der Einfluss weiterer Faktoren auf die Leistung wird ebenfalls beschrieben.

Die Beschreibung der Ergebnisse der verschiedenen Testszenarien wird jeweils eingeleitet mit der Darstellung der erreichten maximalen Benutzerlasten unter Verwendung verschiedener Konfigurationen (Delay-Einstellung und Update-Rate). Darauf folgt jeweils die Beschreibung der Ergebnisse der Messwerte der verschiedenen Leistungsmetriken. Die Beschreibung der einzelnen Leistungsmetriken beginnt jeweils mit der Darstellung von Diagrammen, worauf das Verhalten dann textuell noch genauer beschrieben wird. Die Leistungsmetriken werden in einer Reihenfolge beschrieben, so dass Zusammenhänge zwischen den Leistungsmetriken deutlich werden.

4.4.2 Testszenario Gesamtsystem

In diesem Abschnitt werden die Ergebnisse aus der Leistungsanalyse zum Testszenario *Gesamtsystem* dargestellt und bewertet.

Mit der Durchführung der in Tabelle 4.3.1 definierten Testläufe, konnten die in Abbildung 4.4.2.1 dargestellten maximalen Benutzerlasten (*Peak Loads*), die der Flugdatenserver verarbeiten kann, ermittelt werden. Bei der maximalen Benutzerlast handelt es sich um eine Schwelle, ab welcher weitere Client-Anfragen nur bedingt verarbeitet werden können, da der Flugdatenserver hinsichtlich seiner Ressourcenverwendung komplett ausgelastet ist (CPU-Auslastung zwischen 95% und 100%).

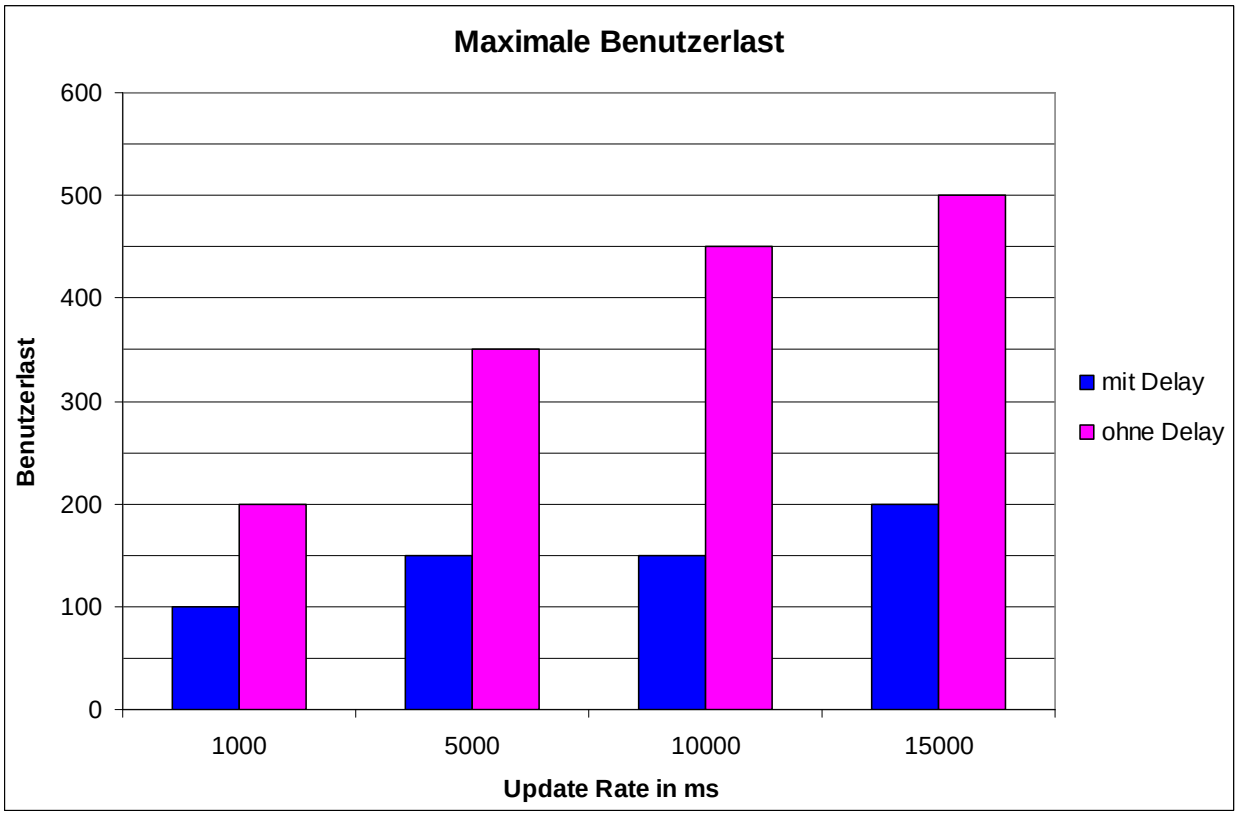


Abb. 4.4.2.1 Maximale Benutzerlasten bei der Analyse des Gesamtsystems.

Wie in der Abbildung 4.4.2.1 dargestellt, können unterschiedliche maximale Benutzerlasten in Abhängigkeit zur Delay-Einstellung (mit oder ohne) und zur Update-Rate vom Flugdatenserver verarbeitet werden. Umso höher die Update-Rate (> 1000 ms) ist, desto größer ist die maximale Benutzerlast, die der Flugdatenserver verarbeiten kann.

Wie einleitend in Abschnitt 4.4.1 erwähnt, sind die Unterschiede der maximalen Benutzerlasten in den beiden Delay-Einstellung auffallend groß (siehe Abbildung 4.4.2.1). Inwiefern dieses Verhalten zustande kommt und auf die in Abschnitt 4.4.1 genannten Unterschiede der Datenbereitstellung zurückzuführen ist, wird nun mit der Betrachtung der Ergebnisse der verschiedenen Leistungsmetriken beschrieben.

Nachfolgend werden die Ergebnisse der Messungen, der in der Tabelle 4.4.1.1 aufgeführten Leistungsmetriken, beschrieben.

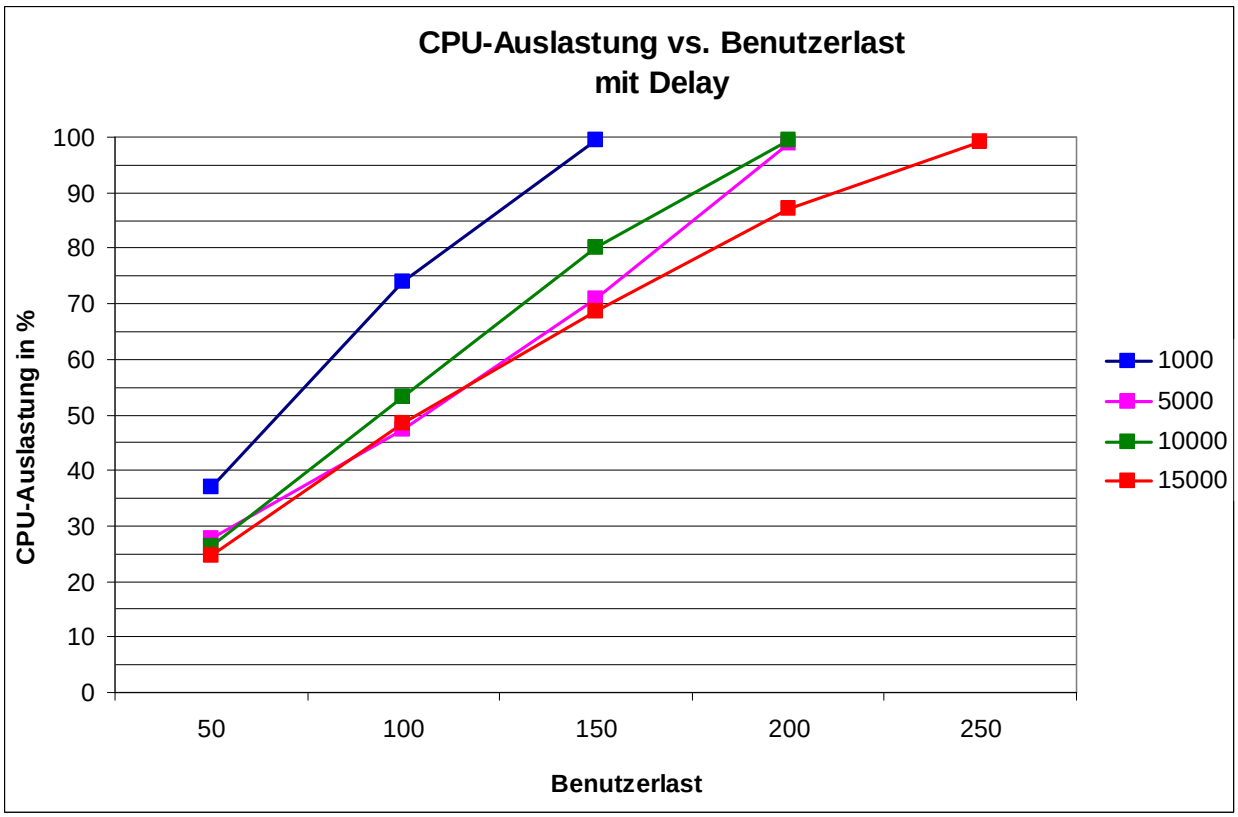


Abb. 4.4.2.2: CPU-Auslastung vs. Benutzerlast mit Delay.

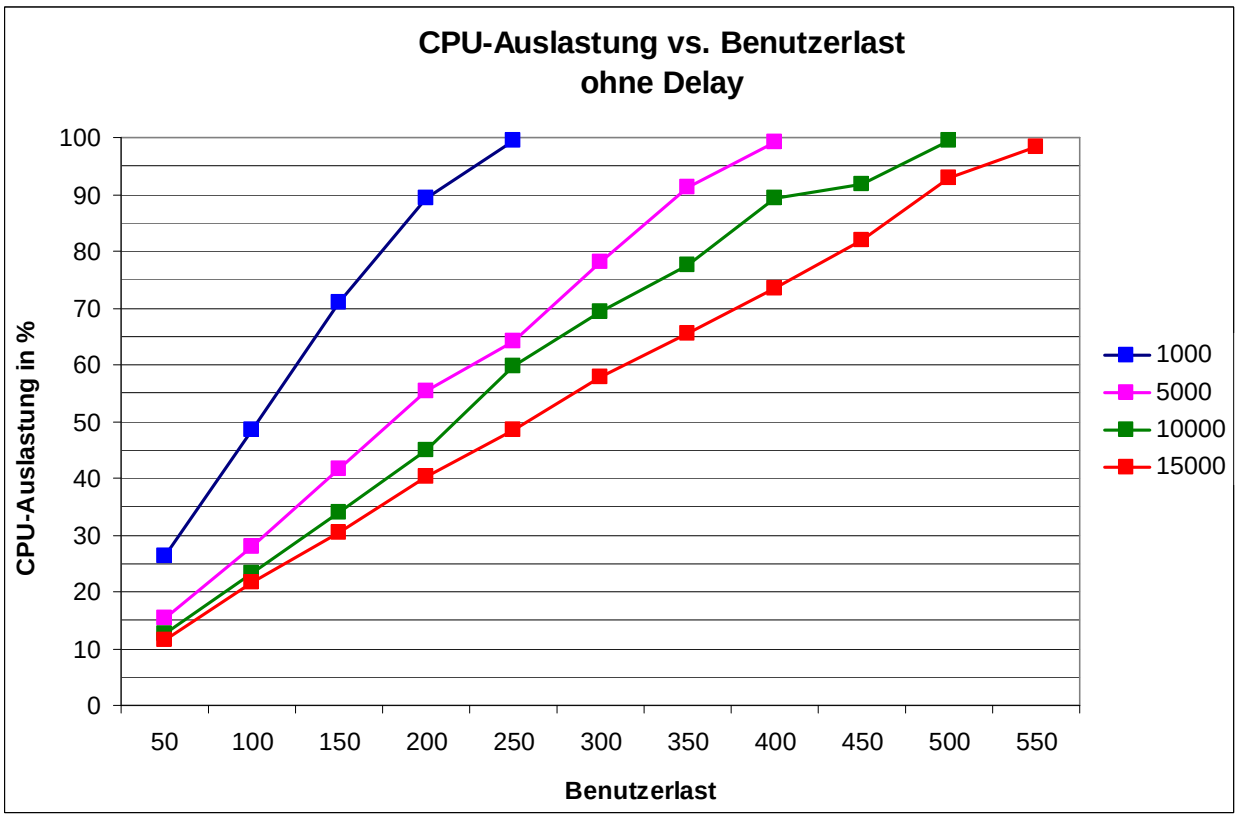


Abb. 4.4.2.3: CPU-Auslastung vs. Benutzerlast ohne Delay.

Vergleich CPU-Auslastung in Prozent					
Update-Rate	**Delay**	**Benutzerlast**			
		50	**100**	**150**	**200**
1000	mit	37,04	74,04		
	ohne	26,34	48,45		
Differenz		10,69	25,59		
5000	mit	27,83	47,34	70,75	
	ohne	15,23	27,96	41,77	
Differenz		12,60	19,38	28,98	
10000	mit	26,32	53,31	80,19	
	ohne	12,63	23,42	34,10	
Differenz		13,69	29,89	46,10	
15000	mit	24,54	48,45	68,76	86,98
	ohne	11,53	21,62	30,48	40,29
Differenz		13,02	26,83	38,27	46,69

Tabelle 4.4.2.1: Vergleich der Messwerte zur CPU-Auslastung vs. Benutzerlast.

- Leistungsmetrik 1: CPU-Auslastung vs. Benutzerlast

In den dargestellten Abbildungen 4.4.2.2 und 4.4.2.3 wird das Verhalten der CPU-Auslastung mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.2.2 wird die CPU-Auslastung *mit Delay* betrachtet und in Abbildung 4.4.2.3 *ohne Delay*. In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtig. In der Tabelle 4.4.2.1 werden zusätzlich die verschiedenen Messreihen aus diesen Abbildungen gegenübergestellt.

In den beiden Abbildungen 4.4.2.2 und 4.4.2.3 und der Tabelle 4.4.2.1 wird deutlich, dass die CPU-Auslastung von der Benutzerlast, der Update-Rate und der Delay-Einstellung abhängig ist.

Die Abhängigkeit der CPU-Auslastung zur Benutzerlast resultiert aus der Menge der Client-Anfragen, die von der Benutzerlast an den Flugdatenserver gestellt werden. Umso größer die Benutzerlast bzw. die Anzahl der Client-Anfragen ist, desto mehr wird die CPU zur Verarbeitung dieser Client-Anfragen beansprucht.

Die Abhängigkeit zur Update-Rate ist in beiden Abbildungen ebenfalls sehr deutlich zu erkennen. Mit größeren Update-Raten (> 1000 ms) wird die CPU deutlich weniger beansprucht, so dass größere Benutzerlasten verarbeitet werden können. So kann mit einer Update-Rate von 15000 ms eine weitaus größere Benutzerlast als mit einer Update-Rate von 1000 ms verarbeitet werden. Die Gründe für dieses Verhalten werden bei Beschreibung der Ergebnisse der nachfolgenden *Leistungsmetrik 2* beschrieben.

Die Abhängigkeit zur Delay-Einstellung wird vor allem in der Tabelle 4.4.2.1 durch den Vergleich der verschiedenen Messwerte zur CPU-Auslastung deutlich. Somit bestehen zwischen den beiden Delay-Einstellung in der CPU-Auslastung, trotz gleicher Update-Rate, erhebliche Unterschiede. Teilweise ist die CPU-Auslastung *mit Delay* doppelt so hoch wie *ohne Delay*. Diese Abhängigkeit wird im Verlauf dieses Abschnittes geklärt.

Weiterhin ist den Abbildungen zu entnehmen, dass die CPU-Auslastung ein eher proportionales Verhalten durch die steigende Benutzerlast zeigt. Der Flugdatenserver kann solange eine steigende Benutzerlast verarbeiten, bis die CPU-Auslastung bei annähernd 100% ist. Diese Schwelle markiert die maximale Benutzerlast (*Peak Load*), die der Flugdatenserver verarbeiten kann. Ist der Flugdatenserver überlastet, kann die bestehende Benutzerlast nur bedingt verarbeitet werden. Inwiefern sich das auswirkt, wird ebenfalls im Verlauf dieses Abschnittes geklärt.

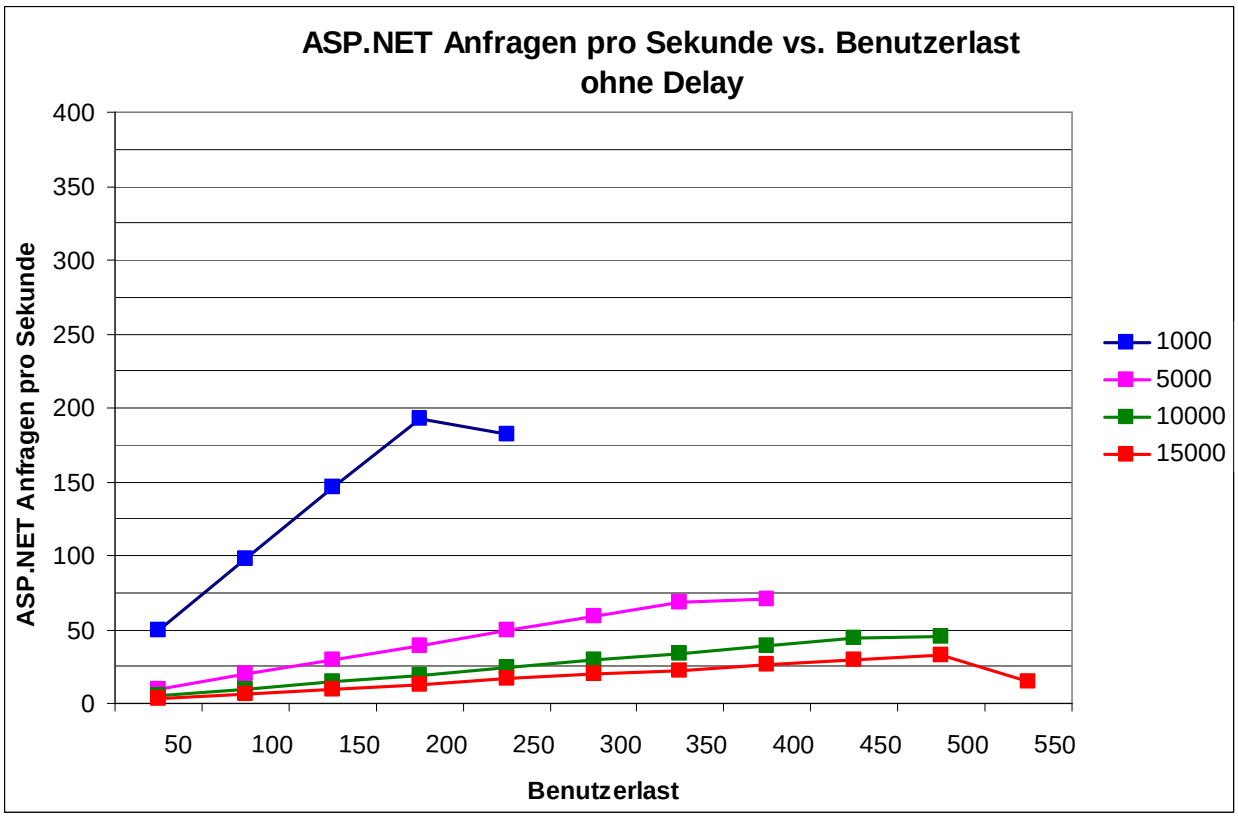


Abb. 4.4.2.4 ASP.NET-Anfragen pro Sekunde vs. Benutzerlast mit Delay.

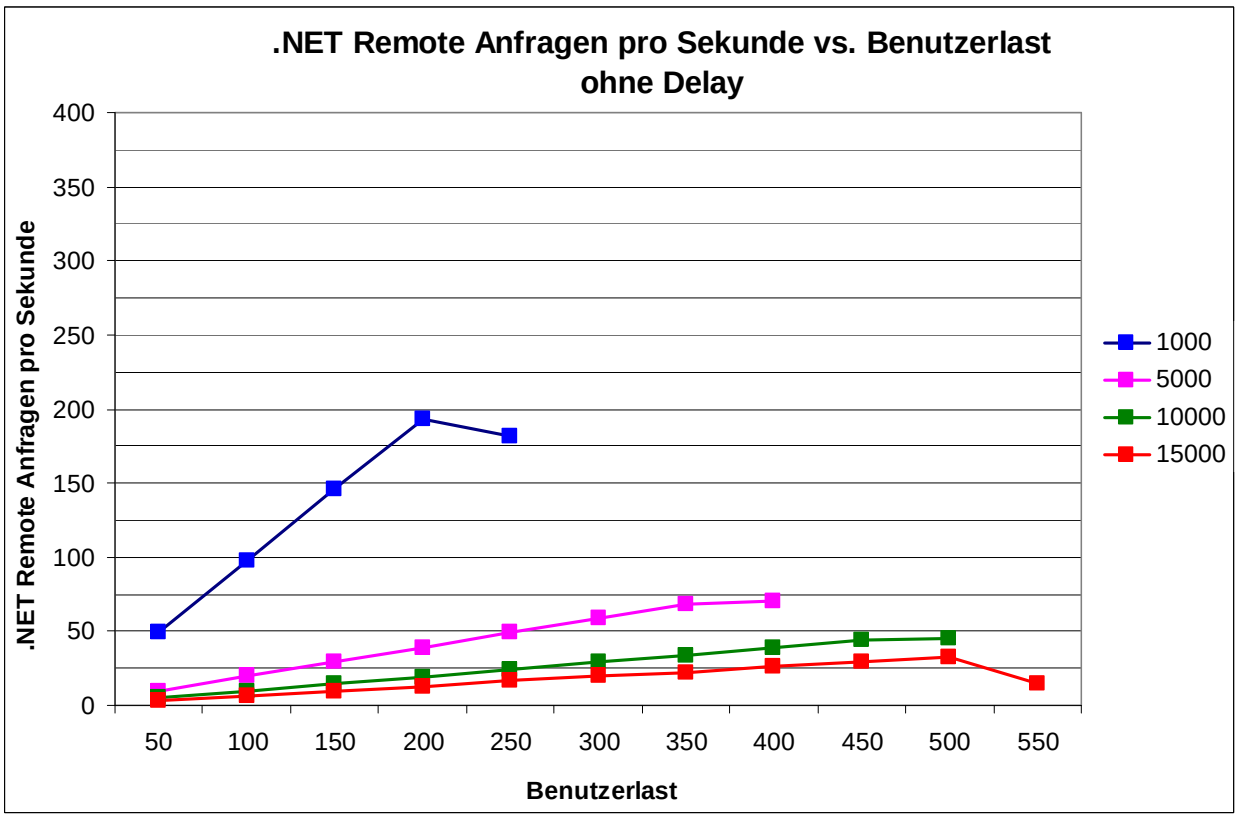


Abb. 4.4.2.5 .NET Remote-Anfragen pro Sekunde vs. Benutzerlast ohne Delay.

- Leistungsmetrik 2: ASP.NET- und .NET Remote-Anfragen pro Sekunde vs. Benutzerlast

In Abbildung 4.4.2.4 wird das Verhalten der verarbeitenden *ASP.NET Anfragen pro Sekunde* mit einer steigenden Benutzerlast dargestellt. In der Abbildung 4.4.2.5 dagegen das Verhalten der *.NET Remote Anfragen pro Sekunde*. Für beide Leistungsmetriken bestehen im Verhalten zu den beiden Delay-Einstellungen (mit oder ohne) keinerlei Unterschiede, so dass stellvertretend jeweils die Messreihen ohne Delay betrachtet werden können. In beiden Abbildungen werden die unterschiedlichen Update-Raten berücksichtigt.

Bei den beiden Leistungsmetriken handelt sich um den Durchsatz an Client-Anfragen den der Flugdatenserver pro Sekunde verarbeitet. Die *ASP.NET Anfragen pro Sekunde* stellen dabei die Client-Anfragen pro Sekunde der simulierten Benutzerlast an den Web-Service auf dem *ATC.Webserver* dar und die *.NET Remote Anfragen pro Sekunde* die Anfragen, die der *ATC.Webserver* an den *ATC.Server* über den *TcpChannel* stellen kann. Wie in Abschnitt 2.2.4 beschrieben, folgt auf eine Web-Service-Anfrage ein RPC vom *ATC.Webserver* zum *ATC.Server*. Somit müsste das Verhalten der beiden Leistungsmetriken in etwa gleich sein. Dies wird in den Abbildungen 4.4.2.4 und 4.4.2.5 bestätigt. Mit den beiden Leistungsmetriken kann also die simulierte Benutzerlast selber überprüft werden.

In den beiden Abbildungen 4.4.2.4 und 4.4.2.5 wird ein proportionales Verhältnis zwischen der Benutzerlast und der Anzahl Anfragen pro Sekunde deutlich. Das proportionale Verhältnis besteht bis zum Erreichen der maximalen Benutzerlast. Danach nehmen die *ASP.NET- und .NET Remote-Anfragen* leicht ab. Dies wird in den beiden Abbildungen am Beispiel einer Update-Rate von 1000 ms deutlich. Bis zu einer Benutzerlast von 200 Benutzern ist das Verhältnis zwischen der Anzahl Anfragen pro Sekunde und der Benutzerlast proportional, danach nehmen die verarbeitenden Anfragen jedoch leicht ab. Damit wird das bereits erwähnte Verhalten bei Überlast des Flugdatenservers bestätigt (siehe Leistungsmetrik 1).

In den Abbildungen wird zudem deutlich, dass die Update-Rate auf die beiden Leistungsmetriken einen großen Einfluss hat. Je größer die Update-Rate (> 1000 ms), desto weniger Client-Anfragen müssen pro Sekunde verarbeitet werden. Dies ergibt sich aus dem Intervall, in dem die Client-Anfragen eines simulierten Benutzers erfolgen. Mit größeren Update-Raten erfolgen die Client-Anfragen eines einzelnen Benutzers dementsprechend in größeren zeitlichen Abständen. In der dazwischen liegenden Zeit kann der Flugdatenserver die Client-Anfragen anderer Benutzer verarbeiten. Dies ist natürlich nur dann gewährleistet, wenn die Client-Anfragen der Benutzerlast nicht zum selben Zeitpunkt erfolgen, was aber sehr

unwahrscheinlich ist. Das unterschiedliche Zeitverhalten wurde bei der Durchführung der *Simulation der Benutzerlast* berücksichtigt. Dieses Verhalten führt dazu, dass mit größeren Update-Raten (> 1000 ms) auch größere Benutzerlasten verarbeitet werden können. Dies ist in den beiden Abbildungen 4.4.2.4 und 4.4.2.5 deutlich zu erkennen. So können mit einer Update-Rate von 15000 ms Client-Anfragen von bis zu 500 Benutzern verarbeitet werden. Mit der Verwendung einer Update-Rate von 1000 ms sind es gerade einmal 200 Benutzer, also nicht einmal die Hälfte.

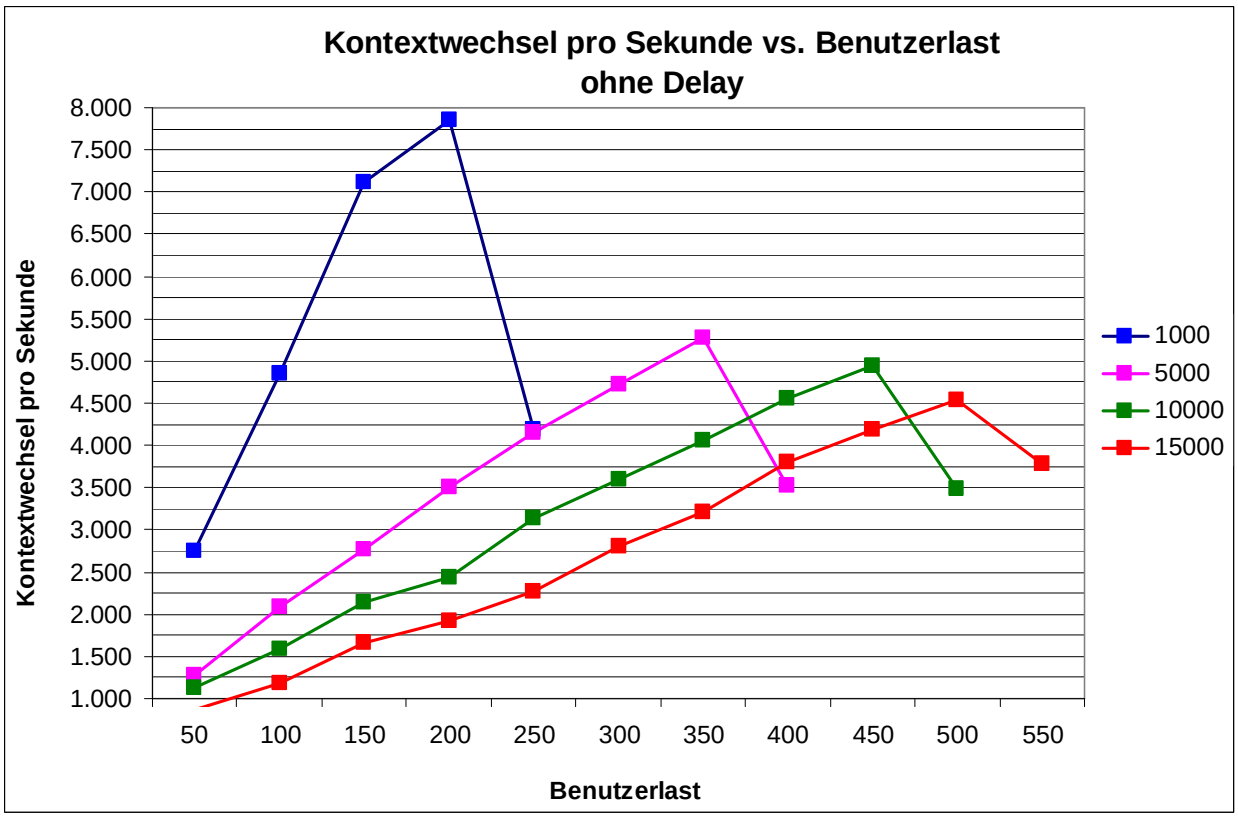


Abb. 4.4.2.6 Kontextwechsel pro Sekunde vs. Benutzerlast ohne Delay.

- Leistungsmetrik 3: Kontextwechsel vs. Benutzerlast

In der Abbildung 4.4.2.6 wird das Verhalten der *Kontextwechsel pro Sekunde* mit einer steigenden Benutzerlast ohne Delay dargestellt. Im Verhalten zu den beiden Delay-Einstellungen (mit und ohne), bestehen hier keinerlei Unterschiede, so dass stellvertretend die *Kontextwechsel pro Sekunde* ohne Delay betrachtet werden können. In der Abbildung werden die unterschiedlichen Update-Raten berücksichtig.

Mit der Leistungsanalyse hat sich herausgestellt, dass mit einer steigenden Benutzerlast in Abhängigkeit zur Update-Rate sehr viele Kontextwechsel pro Sekunde erfolgen. Nachfolgend wird das Verhalten und die Abhängigkeiten erklärt.

In Abbildung 4.4.2.6 wird ein proportionales Verhältnis zwischen der Benutzerlast und den *Kontextwechseln pro Sekunde* bis zu einer bestimmten Schwelle an Benutzern (*Peak Load*) deutlich. Zudem besteht auch hier eine Abhängigkeit zur Update-Rate. Die steigenden

Kontextwechsel pro Sekunde werden durch die eingehenden Client-Anfragen der Benutzerlast verursacht.

Die Web-Service-Anfragen (*Standard- oder Privilegierter-Web-Service*), die an den *ATC.Webserver* gestellt werden, werden durch diesen über ein Multi-Thread-Konzept verarbeitet. Jede eingehende Client-Anfrage wird dabei über einen separaten Thread aus einem *.NET Thread Pool* verarbeitet. Wie bereits in Abschnitt 2.2.4 beschrieben erfolgt innerhalb des Web-Service-Aufrufes ein RPC zum *ATC.Server*. Dieser RPC wird ebenfalls über Multi-Threading im *ATC.Server* verarbeitet. Damit werden für die Verarbeitung einer Client-Anfrage auf dem Flugdatenserver zwei Threads erzeugt. Zum einen für die Verarbeitung der Client-Anfrage im *ATC.Webserver* und zum anderen für den RPC im *ATC.Server*. Dies führt dazu, dass bei einer hohen Anzahl an Client-Anfragen pro Zeiteinheit, z.B. bei einer Update-Rate von 1000 ms, sehr viele Threads gleichzeitig konkurrierend aktiv sind. Eine große Menge an aktiven Threads führt wiederum zu einer erhöhten Anzahl an Kontextwechseln. Generell ist eine hohe Anzahl an Kontextwechseln pro Sekunde aufwendig in der Durchführung und belastet die CPU dementsprechend. Mit höheren Update-Raten (> 1000 ms) gehen weniger Client-Anfragen pro Zeiteinheit (Sekunde) auf dem Flugdatenserver ein (siehe *Leistungsmetrik 2*). Dementsprechend erfolgen auch weniger Kontextwechsel pro Sekunde. Dies ist sicherlich auch ein Faktor, der die Verarbeitung von größeren Benutzerlasten mit höheren Update-Raten ermöglicht.

Das Verhalten bei Überlast des Flugdatenservers wird auch hier bestätigt. Bis zum Erreichen der maximalen Benuterzlast, steigen die *Kontextwechsel pro Sekunde* proportional mit der Benutzerlast. Nach Überschreiten der maximalen Benutzerlast nehmen die *Kontextwechsel pro Sekunde* stark ab, was auf die zurückgehende Anzahl Client-Anfragen bei Überlast des Flugdatenservers zurückzuführen ist.

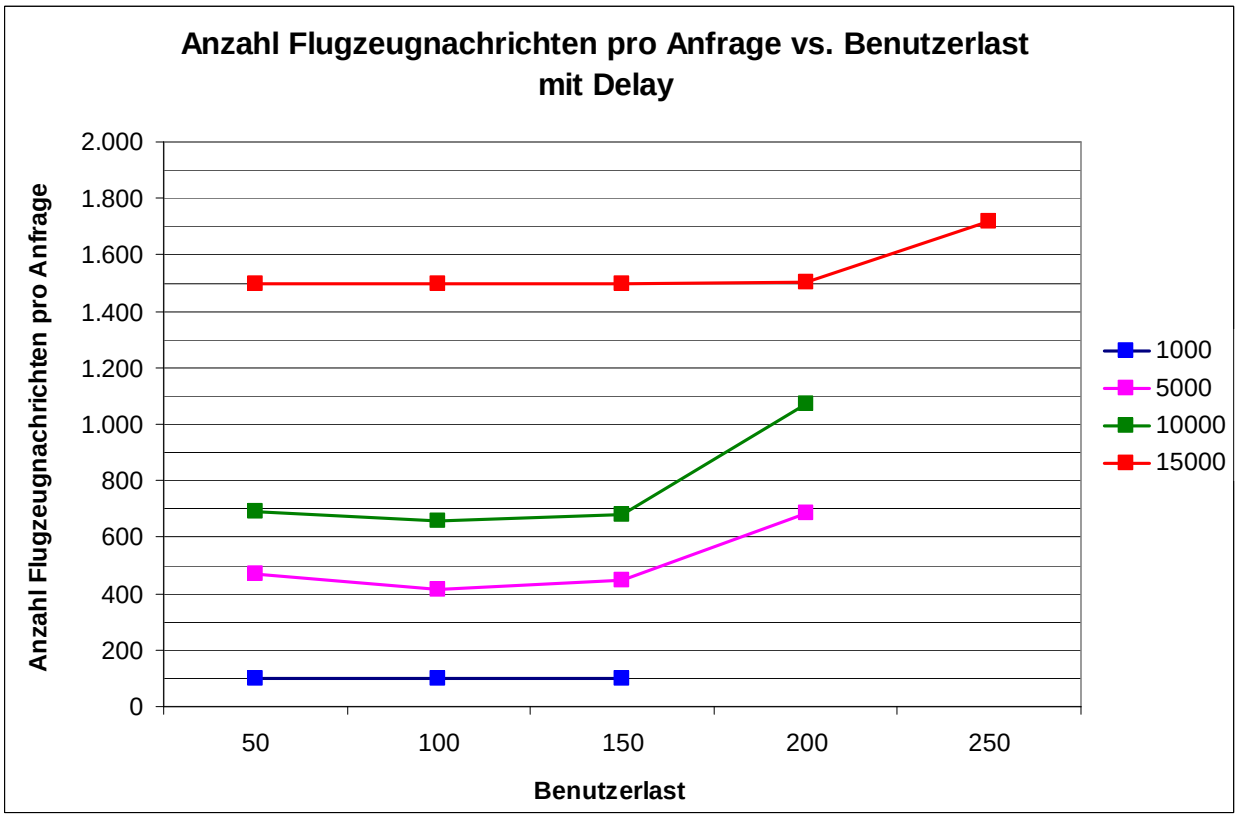


Abb. 4.4.2.7 Anzahl Flugzeugnachrichten pro Anfrage vs. Benutzerlast mit Delay.

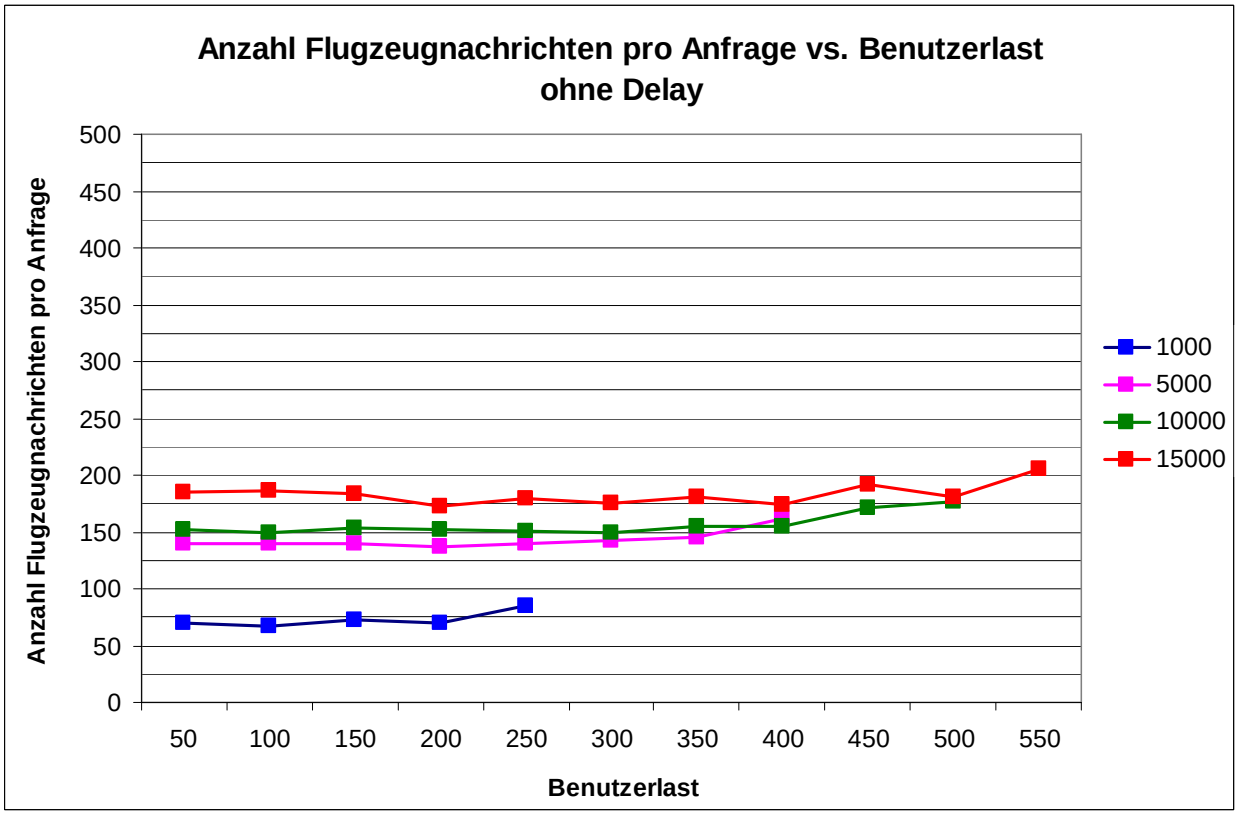


Abb. 4.4.2.8 Anzahl Flugzeugnachrichten pro Anfrage vs. Benutzerlast ohne Delay.

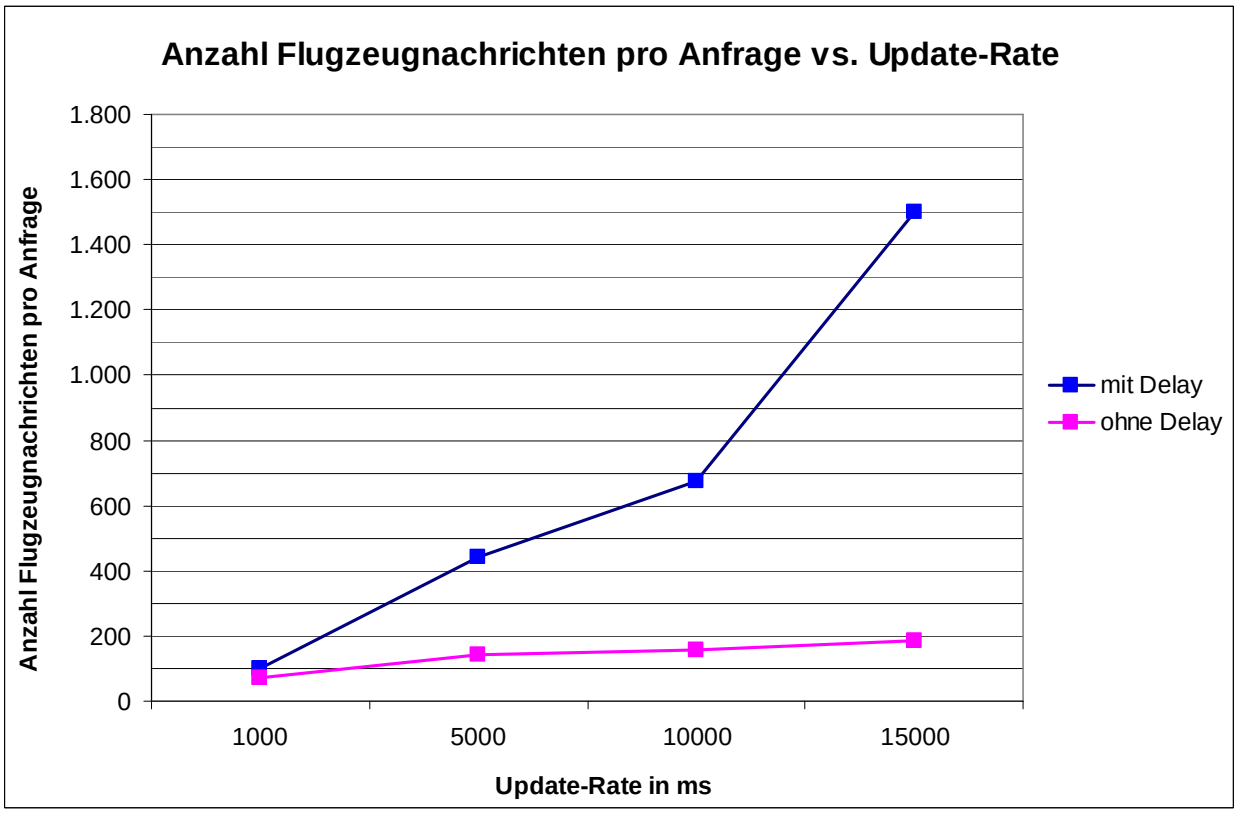


Abb. 4.4.2.9 Anzahl Flugzeugnachrichten pro Anfrage vs. Update-Rate mit und ohne Delay.

- Leistungsmetrik 4: Anzahl Flugzeugnachrichten pro Anfrage vs. Benutzerlast

In den Abbildungen 4.4.2.7 und 4.4.2.8 wird das Verhalten der Anzahl der Flugzeugnachrichten pro Anfrage (Client-Durchsatz) mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.2.7 wird die Anzahl der Flugzeugnachrichten pro Anfrage *mit Delay* und in Abbildung 4.4.2.8 *ohne Delay* betrachtet. In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtigt. In der Abbildung 4.4.2.9 dagegen, wird das Verhalten der Anzahl Flugzeugnachrichten pro Anfrage zur Update-Rate dargestellt, unter Betrachtung der beiden Delay-Einstellungen.

Bei dieser Leistungsmetrik handelt es sich um die Menge an Flugzeugnachrichten die ein Benutzer auf eine Web-Service Anfrage erhält. Nachfolgend wird das Verhalten dieser Leistungsmetrik beschrieben.

In den Abbildungen 4.4.2.7 und 4.4.2.8 wird deutlich, dass keine Abhängigkeit zur Benutzerlast vorhanden ist. Der Flugdatenserver liefert unabhängig von der Benutzerlast immer eine

bestimmte Menge an Flugzeugnachrichten. Diese Menge an Flugzeugnachrichten ist von den verfügbaren ADS-B Antennen bzw. der Menge der Flugzeugnachrichten, die diese Antennen liefern, abhängig. Derzeit versorgen die drei Antennen den Flugdatenserver mit 80 bis 100 Flugzeugnachrichten pro Sekunde.

In beiden Abbildungen 4.4.2.7, 4.4.2.8 und speziell in Abbildung 4.4.2.9 wird jedoch deutlich, dass die Menge an Flugzeugnachrichten pro Anfrage in Abhängigkeit zur Update-Rate und zur Delay-Einstellung steht. Die Abhängigkeit zur Update-Rate ist in den beiden Delay-Einstellungen zudem unterschiedlich ausgeprägt.

In en Abbildungen 4.4.2.7 und 4.4.2.9 ist eine stark ausgeprägte Abhängigkeit zur Update-Rate unter Verwendung des Delays zu erkennen. Umso höher die Update-Rate ist, desto mehr Flugzeugnachrichten werden zurückgegeben. Diese Abhängigkeit ergibt sich aus der angesammelten Menge an Flugzeugnachrichten, die vom Flugdatenserver einer Client-Anfrage bereitgestellt werden kann. Mit einer höheren Update-Rate ist der Zeitraum zwischen zwei Anfragen eines einzelnen Clients entsprechend größer. Bei einer Update-Rate von 15000 ms erfolgen die Anfragen eines Clients demnach alle 15000 ms. In diesem größeren Zeitfenster empfängt der Flugdatenserver mehr Flugzeugnachrichten von den ADS-B Antennen. Bei der bereits erwähnten Eingangsmenge von 80 bis 100 Flugzeugnachrichten pro Sekunde, sammeln sich bei einer Update-Rate von 15000 ms demnach bis zu 1500 Flugzeugnachrichten an, die einer Client-Anfrage zurückgegeben werden können. Dieses Beispiel wird durch die dargestellten Abbildungen 4.4.2.7 und 4.4.2.9 bestätigt.

In der Abbildung 4.4.2.8 dagegen besteht zwar auch eine Abhängigkeit zur Update-Rate, allerdings ist diese nicht so ausgeprägt wie in Abbildung 4.4.2.7. Dieser Unterschied wird in der Abbildung 4.4.2.9 noch einmal deutlich dargestellt. Erhält man auf eine Anfrage *mit Delay* und einer Update-Rate von 15000 ms 1500 Flugzeugnachrichten, so sind es *ohne Delay* gerade einmal 200 Flugzeugnachrichten. Der Unterschied zwischen beiden Delay-Einstellungen resultiert aus der in Abschnitt 2.2.4 und 4.4.1 beschriebenen Optimierungsmaßnahme zur Reduzierung der Flugzeugnachrichten durch Ausfilterung von Mehrfachnachrichten eines Flugzeuges, die im *Privilegierten-Web-Service* (ohne Delay) implementiert ist. In den Abbildungen 4.4.2.8 und 4.4.2.9 wird das Ergebnis dieser Optimierung, gerade bei höheren Updateraten (> 1000 ms), sehr deutlich sichtbar. Im Verlauf dieses Kapitels wird zudem deutlich, inwiefern sich diese Optimierung auf die Messwerte weiterer Leistungsmetriken auswirkt.

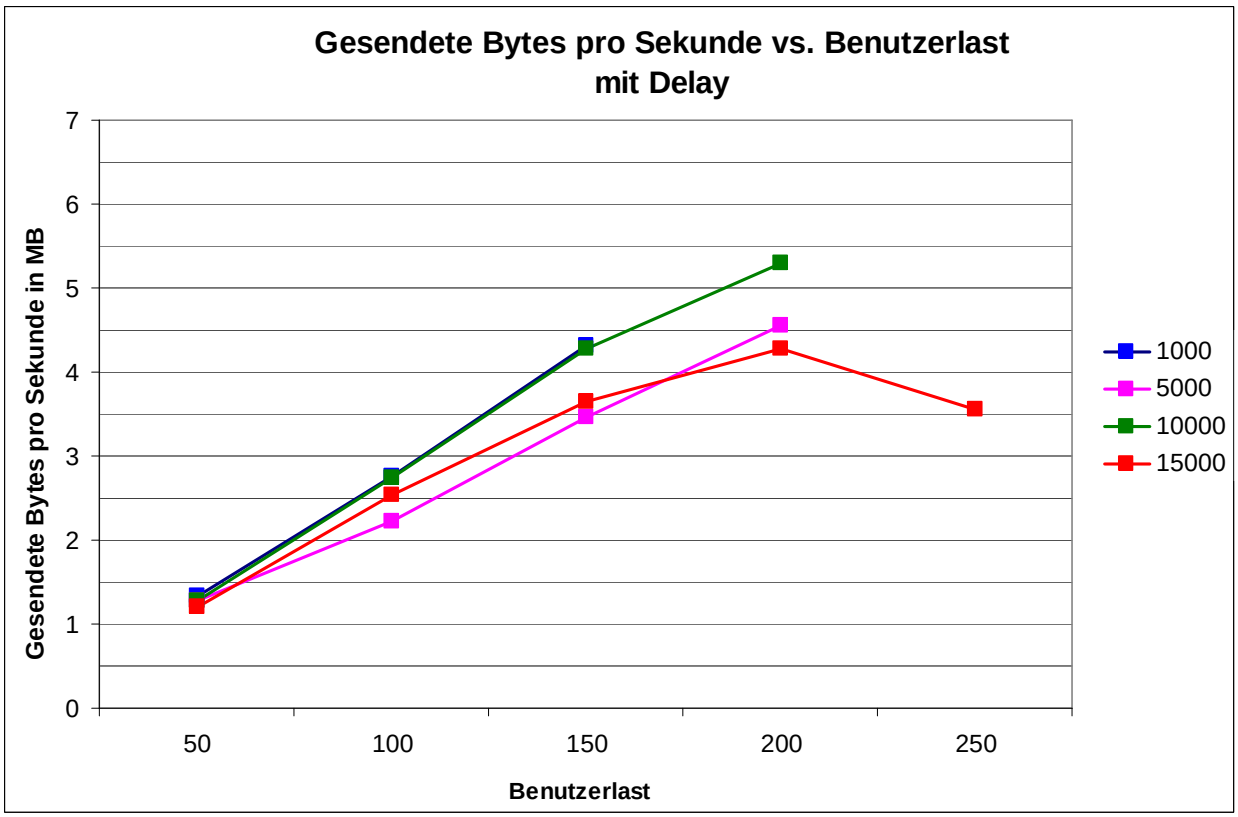


Abb. 4.4.2.10 Gesendete Bytes pro Sekunde vs. Benutzerlast mit Delay.

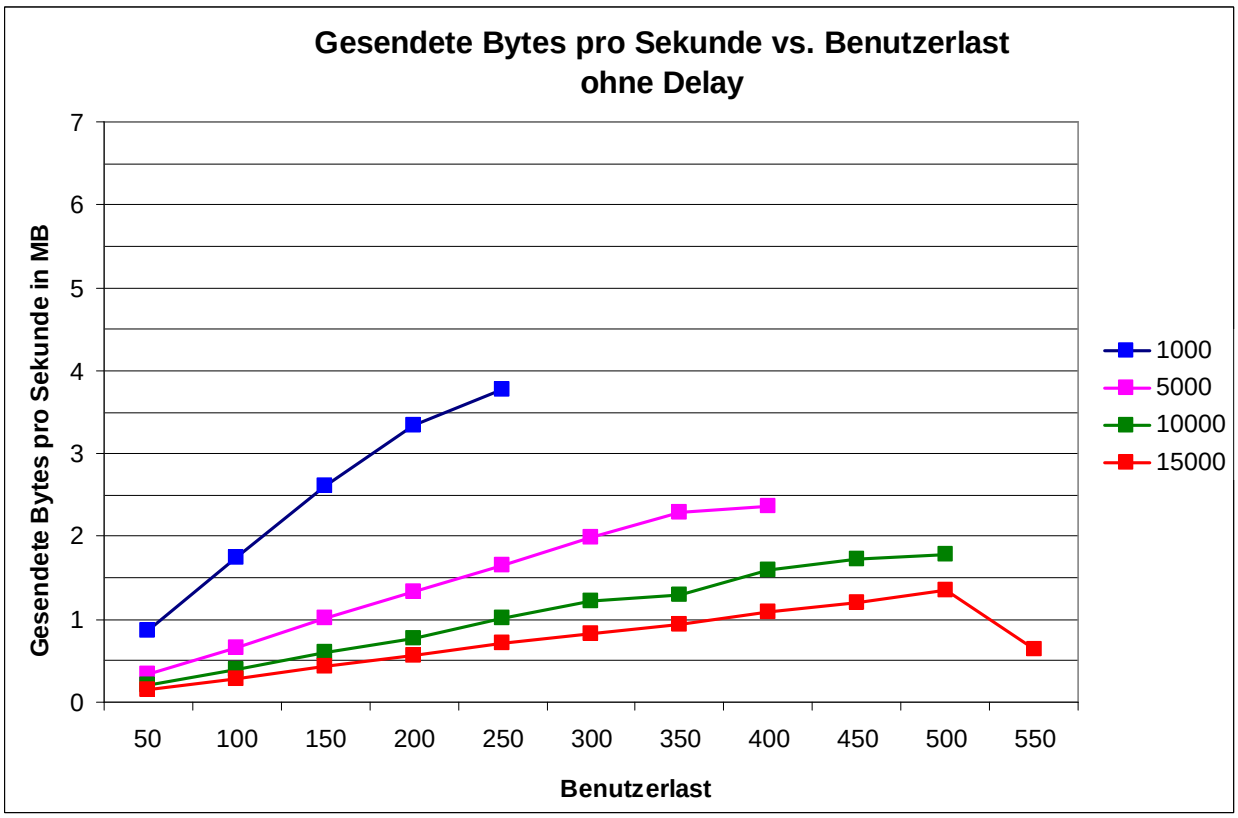


Abb. 4.4.2.11 Gesendete Bytes pro Sekunde vs. Benutzerlast ohne Delay.

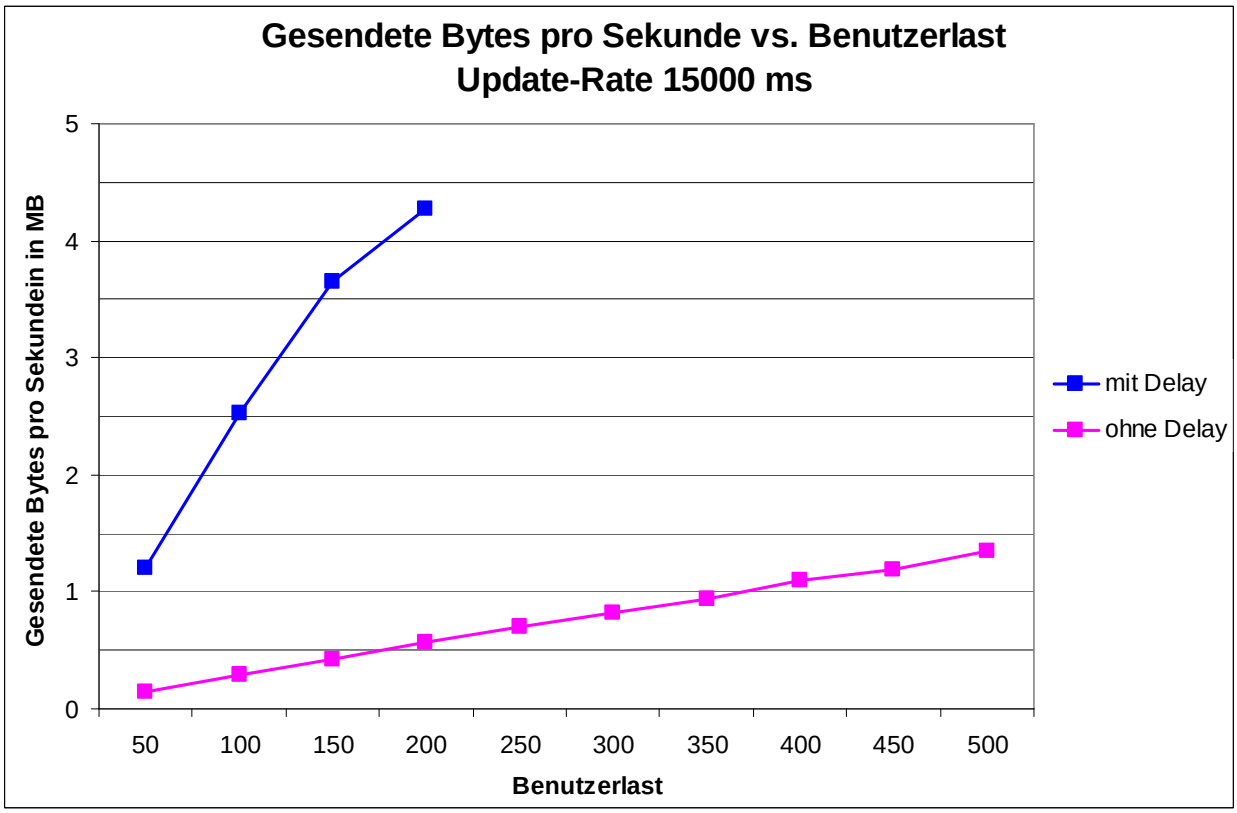


Abb. 4.4.2.12 Gesendete Bytes pro Sekunde vs. Benutzerlast mit und ohne Delay.

- Leistungsmetrik 5: Gesendete Bytes pro Sekunde vs. Benutzerlast

In den Abbildungen 4.4.2.10 und 4.4.2.11 wird das Verhalten der gesendeten Bytes pro Sekunde der Netzwerkschnittstelle des Flugdatenservers mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.2.10 werden die *gesendeten Bytes pro Sekunde mit Delay* betrachtet und in Abbildung 4.4.2.11 *ohne Delay.* In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtigt. In der Abbildung 4.4.2.12 wird das unterschiedliche Verhalten der *Gesendeten Bytes pro Sekunde* zur Delay-Einstellung am Beispiel einer Update-Rate von 15000 ms dargestellt.

Das Verhalten der *Gesendeten Bytes pro Sekunde* spiegelt das *Verhalten der Anzahl Flugzeugnachrichten pro Anfrage* zur Benutzerlast wieder, da die Flugzeugnachrichten, die an einen Web-Client zurückgegeben werden, über die Netzwerkschnittstelle des Flugdatenservers versendet werden.

In Abbildung 4.4.2.10 steigen die gesendeten Bytes eher proportional mit einer steigenden Benutzerlast unabhängig von der verwendeten Update-Rate. Umso größer die Benutzerlast ist, desto mehr Flugzeugnachrichten (Bytes) müssen entsprechend versendet werden. Dabei spielt die Update-Rate in der die Client-Anfragen gestellt werden keine Rolle. Bei einer Update-Rate von 1000 ms erfolgen die Client-Anfragen dementsprechend häufig (jede Sekunde), worauf aber mit einer vergleichsweise geringen Menge an Flugzeugnachrichten geantwortet wird (80 bis 100 Flugzeugnachrichten). Mit höheren Update-Raten (> 1000 ms) finden zwar weniger häufig Client-Anfragen statt, allerdings wird dabei mit einer größeren Menge an Flugzeugnachrichten geantwortet. Mit einer Update-Rate von 15000 ms werden bis zu 1500 Flugzeugnachrichten versendet. Im Verhältnis ist die Menge der Bytes, die bei der Verwendung der unterschiedlichen Update-Raten übertragen werden, jedoch gleich. Wie in Abbildung 4.4.2.10 dargestellt, sind dabei Höchstwerte von bis zu 5,5 MB pro Sekunde möglich.

In Abbildung 4.4.2.11 steigen die gesendeten Bytes ebenfalls mit einer steigenden Benutzerlast. Allerdings steigen die gesendeten Bytes, gerade bei höheren Update-Raten (> 1000 ms), nicht so stark wie das in Abbildung 4.4.2.10 der Fall ist. Dieser Unterschied zwischen den beiden Delay-Einstellungen wird in der Abbildung 4.4.2.12 noch einmal verdeutlicht. Dieses Verhalten resultiert aufgrund der unterschiedlichen Menge an Flugzeugnachrichten, die in den beiden Delay-Einstellungen bereitgestellt werden (*siehe Leistungsmetrik 4*). Umso mehr Flugzeugnachrichten bereitgestellt werden, desto mehr Bytes werden auch versendet. Besonders ausgeprägt ist dieses Verhalten wieder bei höheren Updaterate (10000 und 15000 ms).

Grundsätzlich hat der Flugdatenserver überhaupt keine Probleme die zu versendende Datenmenge zur bewältigen. Dadurch dass der Flugdatenserver mit einer 1 Gbit Netzwerkkarte (128 MB/sek) ausgerüstet ist, stellt der Höchstwert von 5,5 MB/sek überhaupt kein Problem dar. Allerdings wirkt sich die Anzahl der zu sendenden Bytes auf weitere Leistungsmetriken, die im Verlauf dieses Abschnitts noch beschrieben werden, aus.

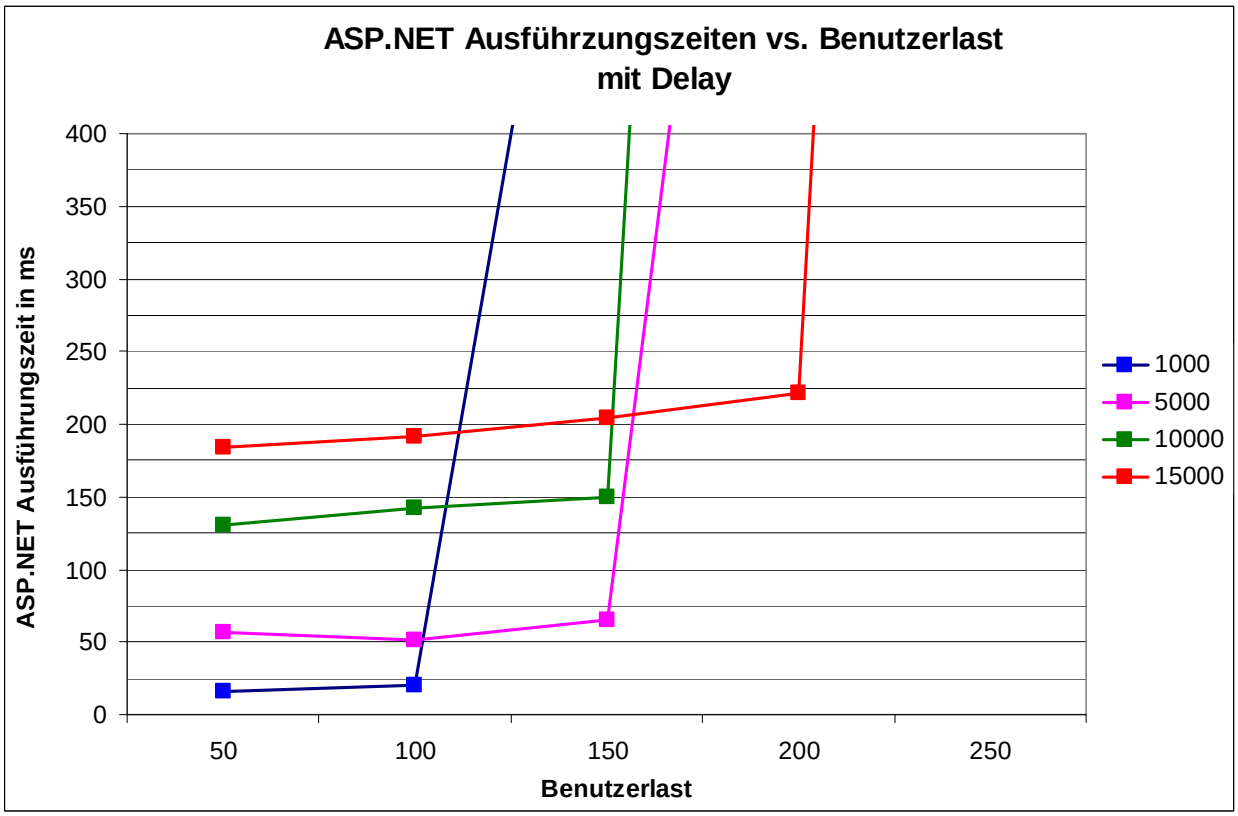


Abb. 4.4.2.13 ASP.NET Ausführungszeiten vs. Benutzerlast mit Delay

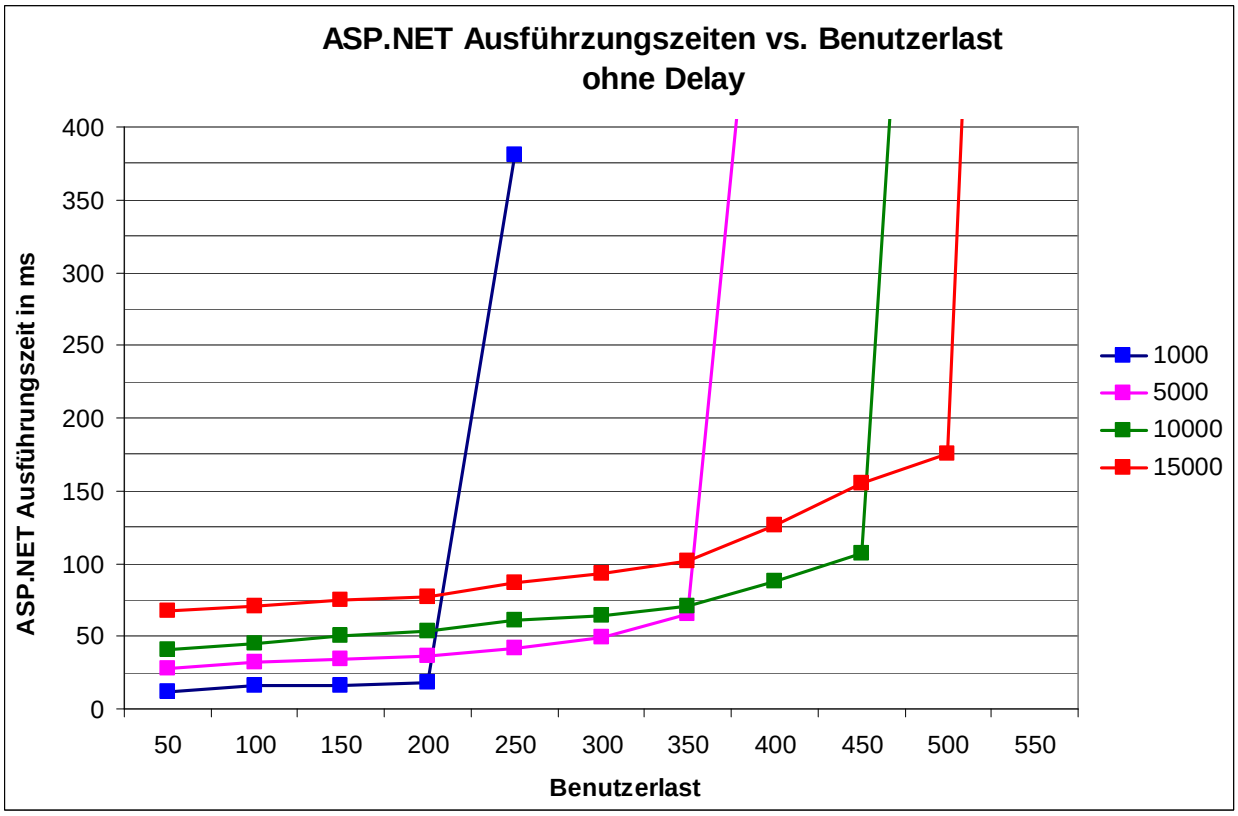


Abb. 4.4.2.14 ASP.NET Ausführungszeiten vs. Benutzerlast ohne Delay

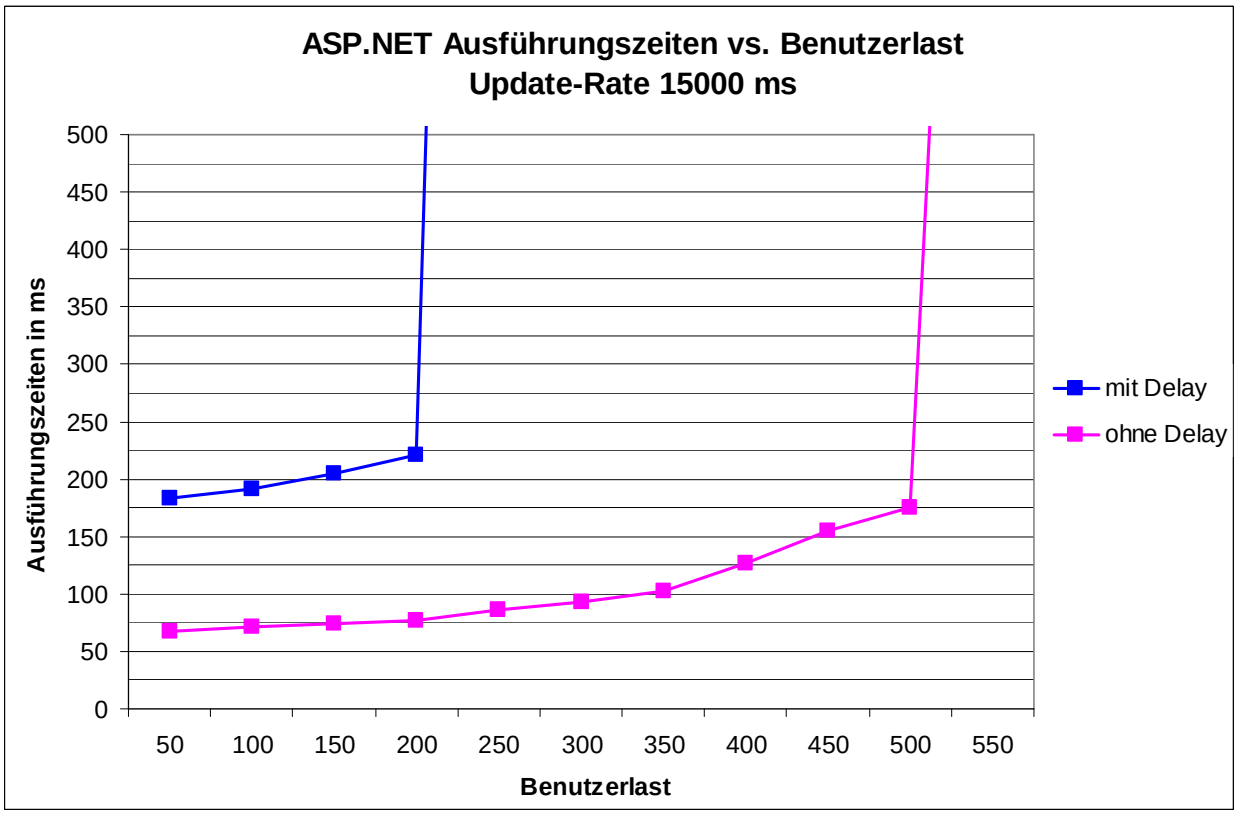


Abb. 4.4.2.15 ASP.NET Ausführungszeiten vs. Benutzerlast mit und ohne Delay

- Leistungsmetrik 6: ASP.NET Ausführungszeiten vs. Benutzerlast

In den Abbildungen 4.4.2.13 und 4.4.2.14 wird das Verhalten der Ausführungszeiten der Client-Anfragen mit einer steigenden Benutzerlast beschrieben. In Abbildung 4.4.2.13 werden die Ausführungszeiten *mit Delay* und in Abbildung 4.4.2.14 *ohne Delay* betrachtet. In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtigt. In der Abbildung 4.4.2.15, wird das unterschiedliche Verhalten der Ausführungszeiten zur Delay-Einstellung am Beispiel einer Update-Rate von 15000 ms dargestellt.

Bei dieser Metrik handelt es sich um die reine Verarbeitungszeit eines Web-Service-Aufrufes (*Standard- oder Privilegierter-Web-Service*) auf dem *ATC.Webserver*. Für die Bewertung der Ausführungszeiten ist es jedoch wichtig, die verschiedenen Aktionen innerhalb des Web-Service zu kennen. Nach der Beschreibung des Ablaufes der Kommunikation in Abschnitt 2.2.4, umfasst die Verarbeitung eines Web-Service-Aufrufes dabei grob die nachfolgenden Aktionen:

- Annahme der HTTP-Anfrage (u.a. Authentifizierung) durch den *ATC.Webserver*.
- *RPC* über den *TcpChannel* vom *ATC.Webserver (Web-Service)* zum *ATC.Server* für die Bereitstellung von Flugzeugnachrichten.
- Reduzierung der bereitgestellten Flugzeugnachrichten (nur im Privilegierten-Web-Service / ohne Delay).
- „Verpacken“ der Flugzeugnachrichten in das JSON-Format für die HTTP-Antwort an den Web-Client.

Grundsätzlich ist in beiden Abbildungen 4.4.2.13 und 4.4.2.14 zu erkennen, dass die Ausführungszeiten mit einer steigenden Anzahl an Benutzern bis zur maximalen Benutzerlast (*Peak Load*) nur leicht ansteigen. Wird die maximale Benutzerlast überschritten, steigen die Ausführungszeiten dann jedoch stark an. Aus Darstellungsgründen sind die Maximalwerte der Ausführungszeiten in den Abbildungen nicht ersichtlich (teilweise bis zu 3 Sekunden). Stellvertretend für dieses Verhalten können die Ausführungszeiten unter Verwendung des Delays und einer Update-Rate von 1000 ms betrachtet werden. Hier liegen die Ausführungszeiten bis zu einem Peak-Load von 100 Benutzern unter 25 ms. Übersteigt die Benutzerlast diesen Peak-Load auf 150 Benutzer, steigen die Ausführungszeiten auf bis zu 700 ms.

Die enorm steigenden Ausführungszeiten sind auf die Überlastung des Flugdatenservers bei Überschreiten der maximalen Benutzerlast zurückzuführen (siehe Leistungsmetrik 1). Wie schon erwähnt ist die CPU-Auslastung bei Überschreiten der maximalen Benutzerlast bei annähernd 100%, was dazu führt, dass sich für die Verarbeitung der Client-Anfragen auf dem Flugdatenserver längere Wartezeiten ergeben. Im Prinzip lässt sich dieses Verhalten mit mehreren konkurrierenden Threads auf einem System vergleichen. Desto mehr Threads auf einem System aktiv sind, desto länger werden die Wartezeiten der einzelnen Threads, die sich durch das Scheduling des Betriebsystems ergeben. Durch die längeren Wartezeiten zur Verarbeitung der Client-Anfragen, erhöhen sich automatisch die Verarbeitungszeiten der Client-Anfragen auf dem *ATC.Webserver.*

In den beiden Abbildungen 4.4.2.13 und 4.4.2.14 und speziell in Abbildung 4.4.2.15 ist zudem ein unterschiedliches Niveau der Ausführungszeiten in den beiden Delay-Einstellungen trotz

gleicher Update-Rate (5000 ms, 10000 ms und 15000 ms) zu erkennen. Liegen die Ausführungszeiten unter Verwendung des Delays und einer Update-Rate von 15000 ms, zwischen 175 und 200 ms, so liegen diese ohne Delay und unter Verwendung derselben Update-Rate, gerade einmal zwischen 50 und 75 ms. Dies ist immerhin ein Unterschied von 125 ms.

Dieses unterschiedliche Niveau der Ausführungszeiten zwischen den beiden Delay-Einstellungen bei gleicher und hoher Update-Rate (> 1000 ms) resultiert aus der unterschiedlichen Menge an Flugzeugnachrichten, die in Abhängigkeit zur Delay-Einstellung und Update-Rate einer Client-Anfrage bereitgestellt werden (siehe Leistungsmetrik 4). Werden unter Verwendung des Delays und einer Update-Rate von 15000 ms derzeit bis zu 1500 Flugzeugnachrichten dem Web-Client bereitgestellt, sind es *ohne Delay* gerade einmal 200 Flugzeugnachrichten. Diese Menge an Flugzeugnachrichten, die dem *ATC.Webserver* zur Rückgabe an die Web-Clients bereitgestellt werden, beeinflussen maßgeblich die Ausführungszeiten, da diese für die HTTP-Antwort serialisiert werden müssen.

Wie einleitend zur Beschreibung dieser Leistungsmetrik erwähnt, werden die Flugzeugnachrichten im JSON-Fomat über die HTTP-Antwort an den Web-Client zurückgegeben. Hierfür wird die zuvor vom *ATC.Server* erhaltene Menge an Flugzeugnachrichten in das JSON-Format umgewandelt (serialisiert) und in das HTTP-Protokoll „verpackt“. Hierbei hat sich herausgestellt, dass die Serialisierung in das JSON-Format weitaus mehr Ressourcen kostet, als im Vergleich zu einer Umwandlung in ein XML-Format. Laut der Quelle [DEV01] wird mit XML zwar ein 20-40% größeren Daten-Overhead erzeugt, allerdings ist die Serialisierung in XML 20-25% schneller als in JSON. Dieser Umstand macht sich gerade bei größeren Mengen an Flugzeugnachrichten stark in der Serialisierungszeit bemerkbar. Die Ausführungszeiten sind demnach von der Anzahl der Flugzeugnachrichten, die zurückgegeben werden sollen, stark abhängig. Damit wird die Ausführungszeit indirekt durch die Optimierungsmaßnahme im *Privilegierten-Web-Service* beeinflusst.

Die Unterschiede bei der Serialisierung zwischen dem XML- und JSON-Format kann exemplarisch aus der nachfolgenden Tabelle 4.4.2.2. entnommen werden. Die darin enthaltenen Ergebnisse wurden in einem separaten Test ermittelt.

Anzahl Flugzeug-nachrichten	XML-Format		JSON-Format	
	Serialisierungs-zeit in ms	Größe in Bytes	Serialisierungs-zeit in ms	Größe in Bytes
100	8	45467	14	25191
500	41	227067	65	126391
700	57	363267	89	202291
1500	118	681567	159	379891

Tabelle 4.4.2.2 Vergleich der Serialisierung in das XML- und JSON-Format.

Zusätzlich besteht eine Abhängigkeit zur Antwortzeit des RPC's zwischen dem *ATC.Webserver* und dem *ATC.Server*. Inwiefern sich diese Unterschiede zwischen den beiden Delay-Einstellungen auf die Ausführungszeiten auswirken, kann hier nicht ermittelt werden. Bei der Betrachtung der Ergebnisse zum Testszenario *Teilsystem* wird jedoch auf diesen Einfluss noch einmal eingegangen.

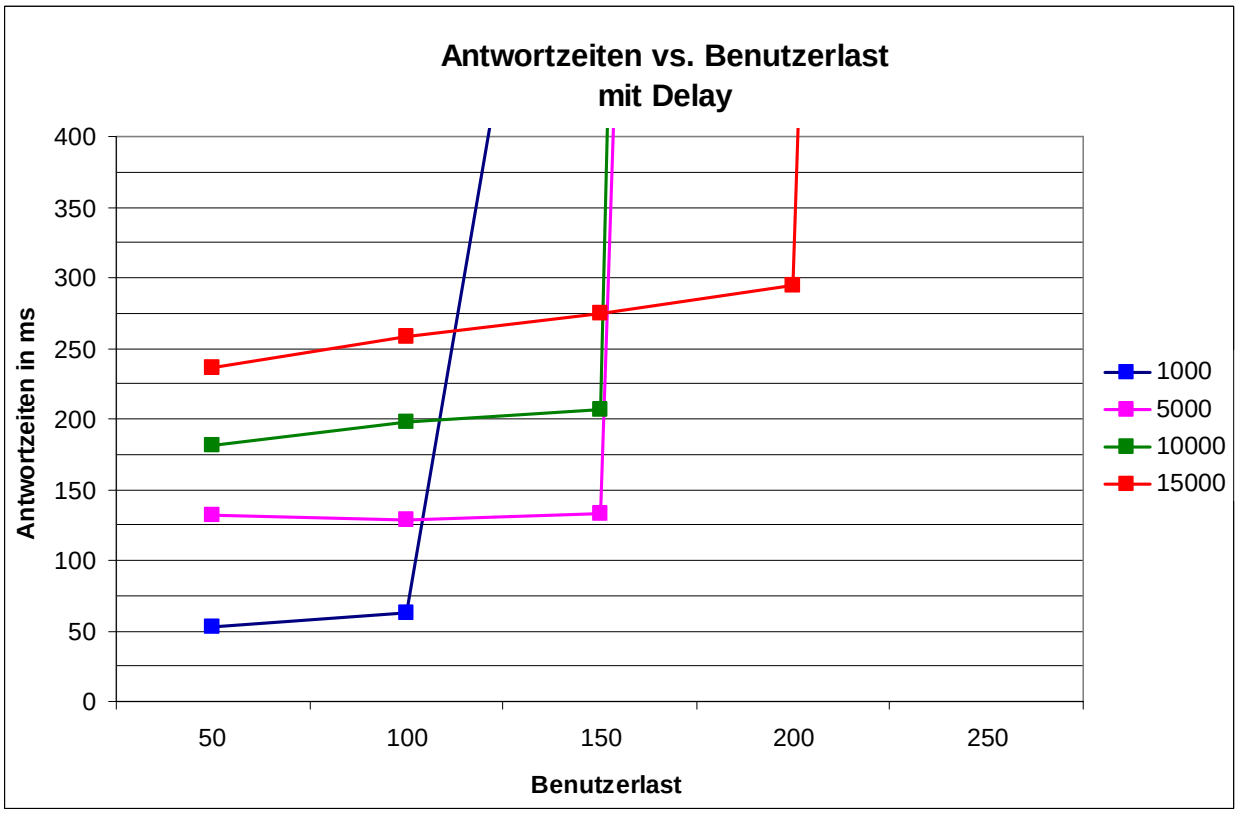


Abb. 4.4.2.16 Antwortzeiten vs. Benutzerlast ohne Delay.

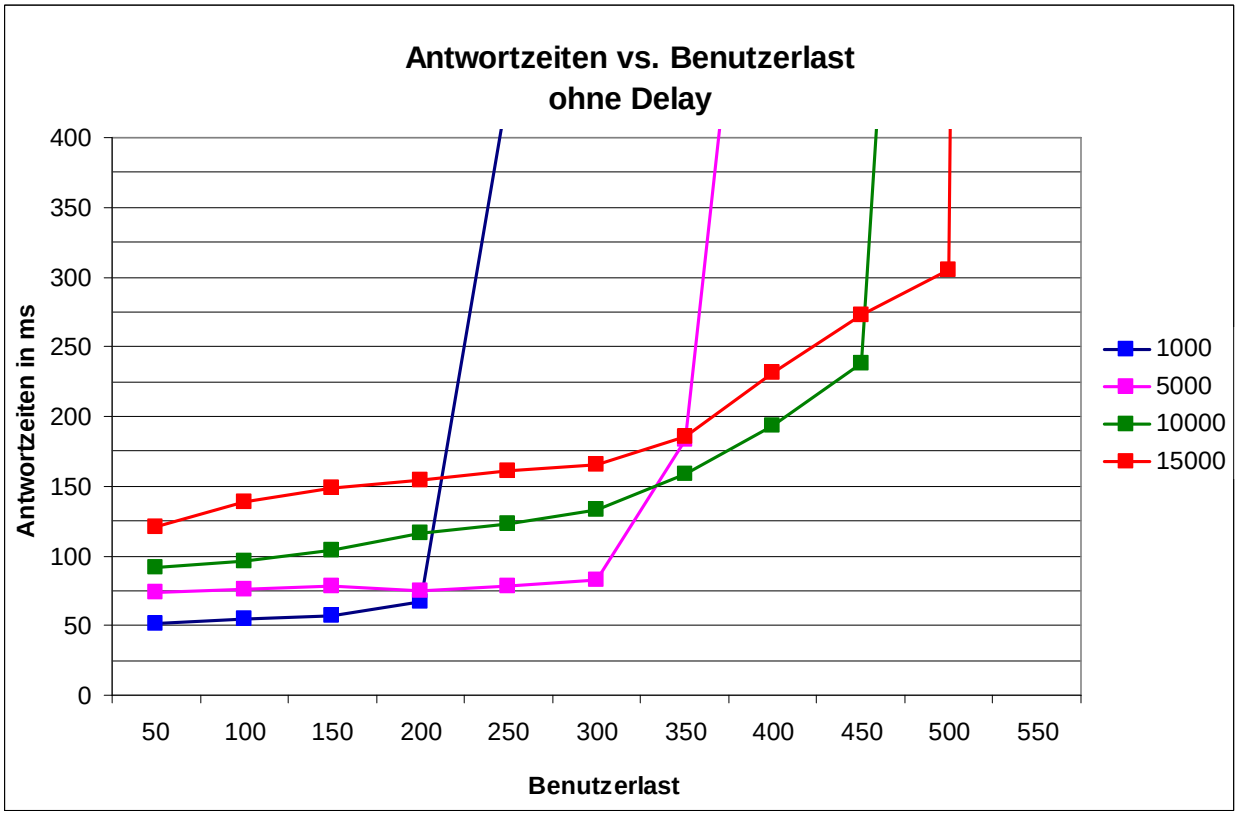


Abb. 4.4.2.17 Antwortzeiten vs. Benutzerlast ohne Delay.

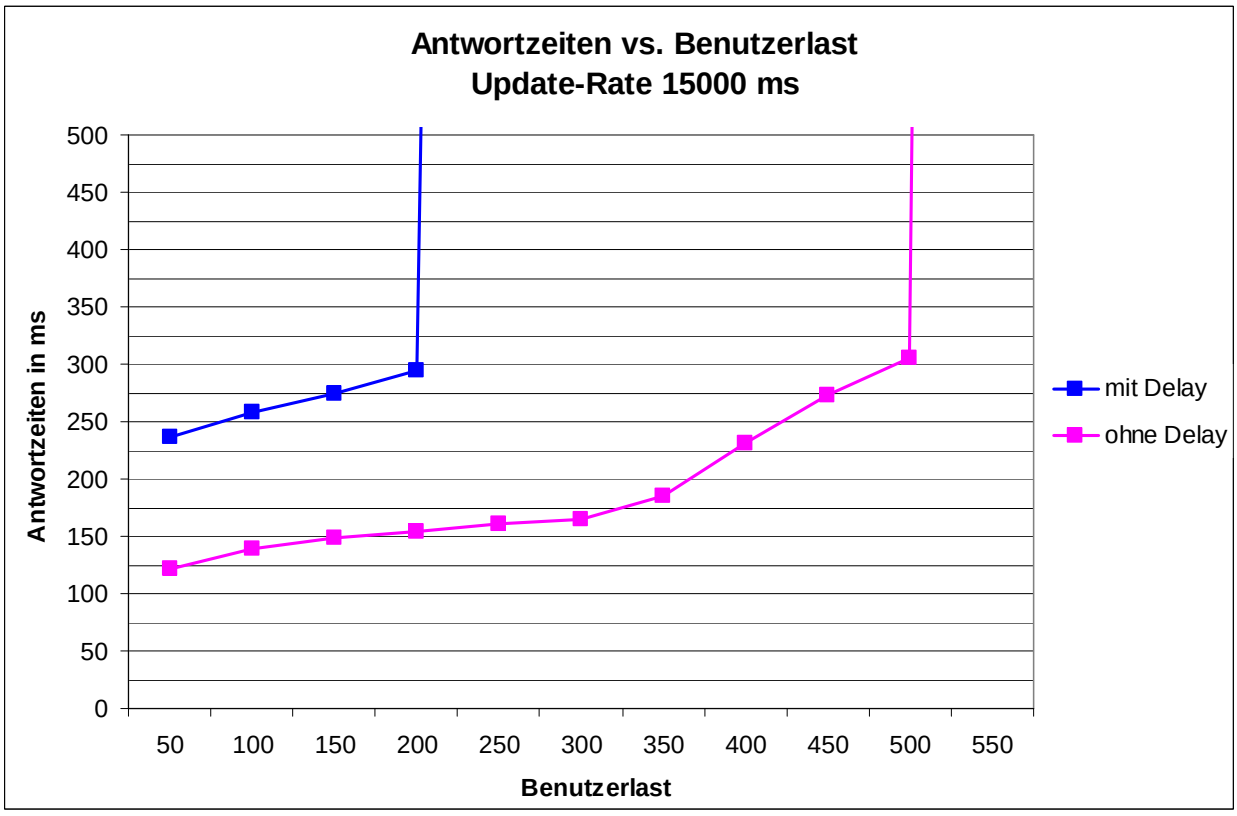


Abb. 4.4.2.18 Antwortzeiten vs. Benutzerlast mit und ohne Delay.

- Leistungsmetrik 7: Antwortzeiten vs. Benutzerlast

In den Abbildungen 4.4.2.16 und 4.4.2.17 wird das Verhalten der Antwortzeiten auf die Client-Anfragen mit einer steigenden Benutzerlast beschrieben. In Abbildung 4.4.2.16 werden die Antwortzeiten *mit Delay* und in Abbildung 4.4.2.17 *ohne Delay* betrachtet. In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtigt. In der Abbildung 4.4.2.18 wird das unterschiedliche Verhalten der Antwortzeiten zur Delay-Einstellung am Beispiel einer Updaterate von 15000 ms dargestellt.

Die Antwortzeit beinhaltet, wie in Abschnitt 2.3.2 beschrieben, den gesamten Zeitraum beginnend mit der Client-Anfrage (Request) bis zum Erhalt der Server-Antwort (Response). Dieser Zeitraum beinhaltet somit die Übertragungszeit über das Netzwerk (Request und Response) sowie die Ausführungszeit der Client-Anfrage auf dem Flugdatenserver (*ATC.Webserver*).

Das Verhalten der Antwortzeiten mit einer steigenden Benutzerlast ist somit äquivalent zum Verhalten der Ausführungszeiten auf dem *ATC.Webserver*. Wie bei den Ausführungszeiten steigen die Antwortzeiten mit einer steigenden Benutzerlast nur leicht an. Wird die maximale Benutzerlast überschritten, steigen die Antwortzeiten aufgrund der enorm steigenden Verarbeitungszeit ebenfalls stark an. Das unterschiedliche Niveau der Antwortzeiten ist ebenfalls auf das Verhalten der verlängerten Verarbeitungszeiten auf dem Flugdatenserver zurückzuführen (siehe Leistungsmetrik 6).

Zusätzlich besteht jedoch noch eine starke Abhängigkeit zum Netzwerk bzw. zum Übertragungsmedium, mit dem der Web-Client mit dem Flugdatenserver kommuniziert. Abhängig von der zu versendenden Datenmenge sowie der Bandbreite und Auslastung (konkurrierende Zugriffe) im Netzwerk selber, ergeben sich entsprechende unvermeidbare Latenzen bei der Übertragung für die Anfrage vom Web-Client und der Antwort vom *ATC.Webserver*. Nachfolgend ein Vergleich der diese Abhängigkeit deutlich macht.

Mit der Simulation von einem Benutzer mit Delay und einer Update-Rate von 1000 ms innerhalb des Hochschul-Netzwerkes liegen die Antwortzeiten bei 25 bis 30 ms. Mit Ausführung der Simulation von zu Hause über einen 6000 kbit Internetzugang dagegen, liegen die Antwortzeiten bei 350 ms bis 400 ms.

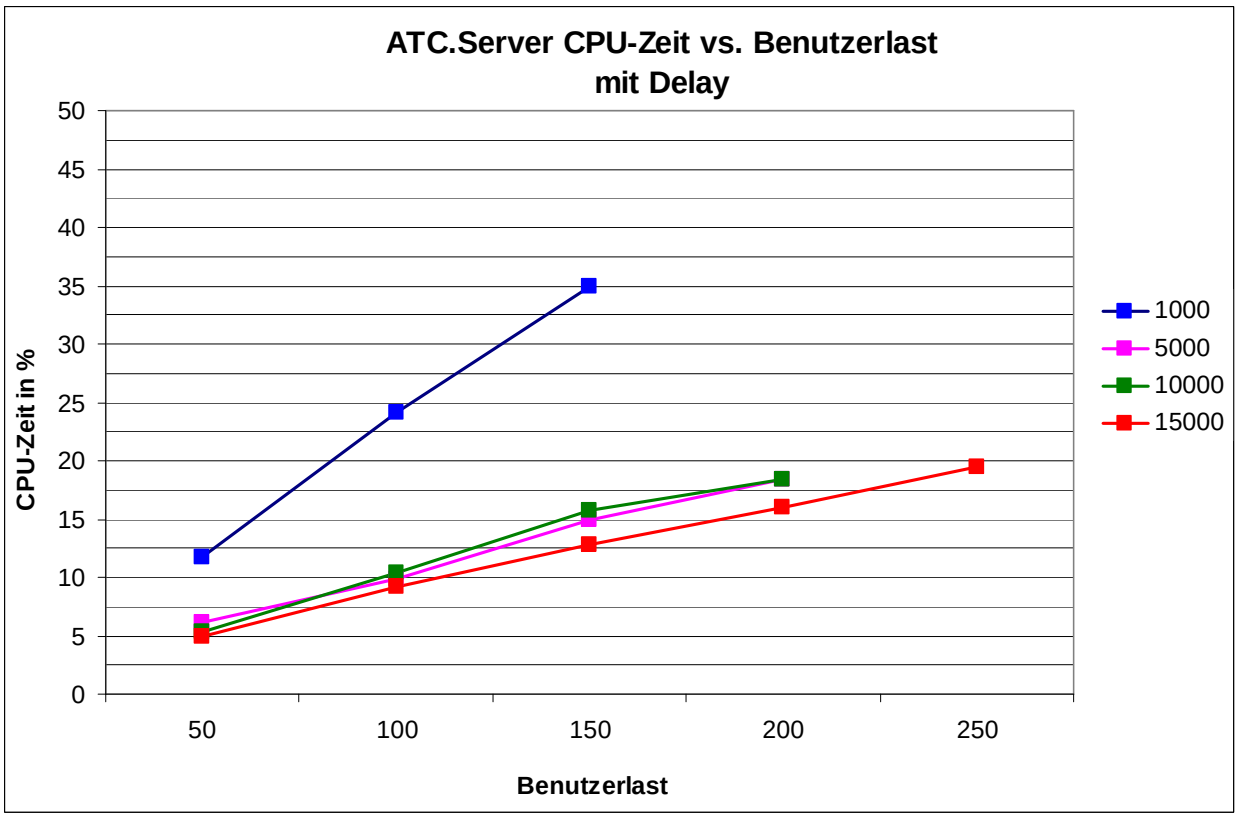


Abb. 4.4.2.19: ATC.Server CPU-Zeit vs. Benutzerlast mit Delay.

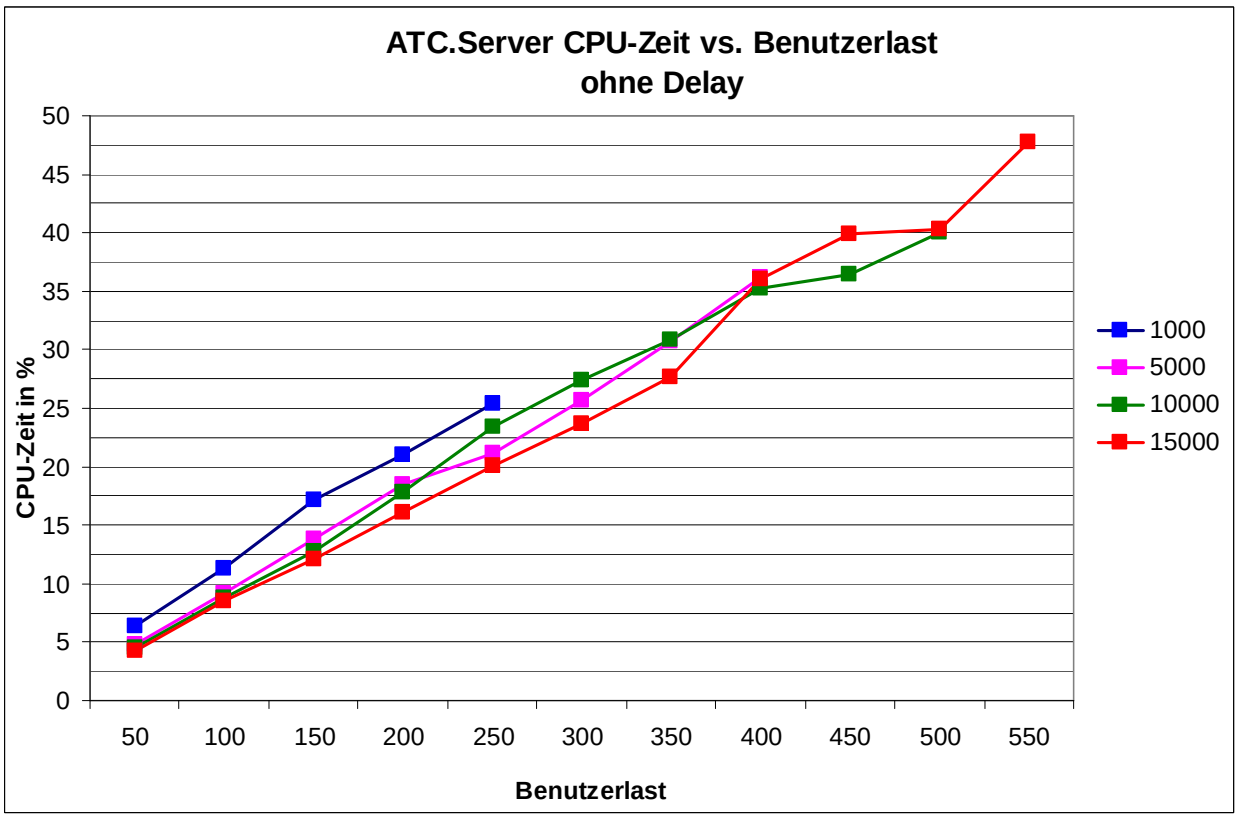


Abb. 4.4.2.20: ATC.Server CPU-Zeit vs. Benutzerlast ohne Delay.

Vergleich ATC.Server CPU-Zeit in Prozent					
Update-Rate	**Delay**	**Benutzerlast**			
		50	**100**	**150**	**200**
1000	mit	11,70	24,11		
	ohne	6,33	11,24		
Differenz		5,37	12,87		
5000	mit	6,17	9,87	14,95	
	ohne	4,76	9,14	13,82	
Differenz		1,41	0,73	1,13	
10000	mit	5,30	10,39	15,67	
	ohne	4,52	8,77	12,81	
Differenz		0,78	1,62	2,86	
15000	mit	4,95	9,19	12,80	15,99
	ohne	4,26	8,50	12,06	16,12
Differenz		0,69	0,70	0,74	-0,13

Tabelle 4.4.2.3: Vergleich der Messwerte zur CPU-Zeit der ATC.Server vs. Benutzerlast.

- Leistungsmetrik 8: ATC.Server CPU-Zeit vs. Benutzer

In den dargestellten Abbildungen 4.4.2.19 und 4.4.2.20 wird das Verhalten der verwendeten CPU-Zeit des *ATC.Servers* mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.2.19 wird die verwendete CPU-Zeit *mit Delay* betrachtet und in Abbildung 4.4.2.20 *ohne Delay*. Zudem werden in beiden Abbildungen die unterschiedlichen Update-Raten berücksichtig. In der Tabelle 4.4.2.3 werden zusätzlich die verschiedenen Messreihen gegenübergestellt.

Aus den beiden Abbildungen 4.4.2.19 und 4.4.2.20 und der Tabelle 4.4.2.3 geht hervor, dass der Anteil der verwendeten CPU-Zeit, sich in etwa proportional zur Benutzerlast verhält. Der *ATC.Server* ist somit recht skalierbar.

Auffallend ist jedoch der Unterschied der verwendeten CPU-Zeiten zwischen den beiden Delay-Einstellungen (mit und ohne Delay) bei Verwendung einer Update-Rate von 1000 ms. Dieser Unterschied wird in der Tabelle 4.4.2.3 noch einmal sehr deutlich dargestellt. Zudem wird in der

Tabelle deutlich, dass der Unterschied in Abhängigkeit zur Benutzerlast steht. Umso größer die Benutzerlast ist, desto größer fällt der Unterschied der verwendeten CPU-Zeit bei gleicher Benutzerlast aus. Leider ist die Ursache für dieses Verhalten mit Hilfe der Messwerte der verschiedenen Leistungsmetriken nicht ermittelbar. Ohne Verwendung eines hinreichend guten *Profiling Tools* kann der Grund für dieses Verhalten nicht exakt bestimmt werden.

Vermutlich resultiert dieser Unterschied aber aufgrund der unterschiedlichen Implementierung für die Datenbereitstellung *mit und ohne Delay* im *ATC.Server*. Wie bereits in Abschnitt 2.2.4 und 4.4.1 beschrieben, muss bei jeder Anfrage *eines Standard-Web-Clients*, für den das Delay berücksichtigt werden muss, die Delay-Position durch Vergleichen von Zeitstempeln der Flugzeugnachrichten berechnet werden. Beim *Privilegierten-Web-Client* (ohne Delay) ist diese Berechnung nicht notwendig und fällt daher weg. Die Unterschiede der Implementierung macht sich vermutlich durch die hohe Anzahl an Client-Anfragen (ASP.NET- und .NET Remote-Anfragen) pro Zeiteinheit (Sekunde) und einer großen Menge an Kontextwechseln, die bis zu einer Update-Rate von 1000 ms entstehen, besonders bemerkbar. Bei der Verwendung von höheren Update-Raten (> 1000 ms) sind die Client-Anfragen und Kontextwechsel pro Zeiteinheit (Sekunde) weitaus geringer, so dass sich der Unterschied bei einer längerer Laufzeit der beiden Implementierungen kaum bemerkbar macht. Mit der Beschreibung der Ergebnisse zum Testszenario *Teilsystem* kann diese Vermutung im Verlauf dieses Kapitels eventuell bestätigt werden.

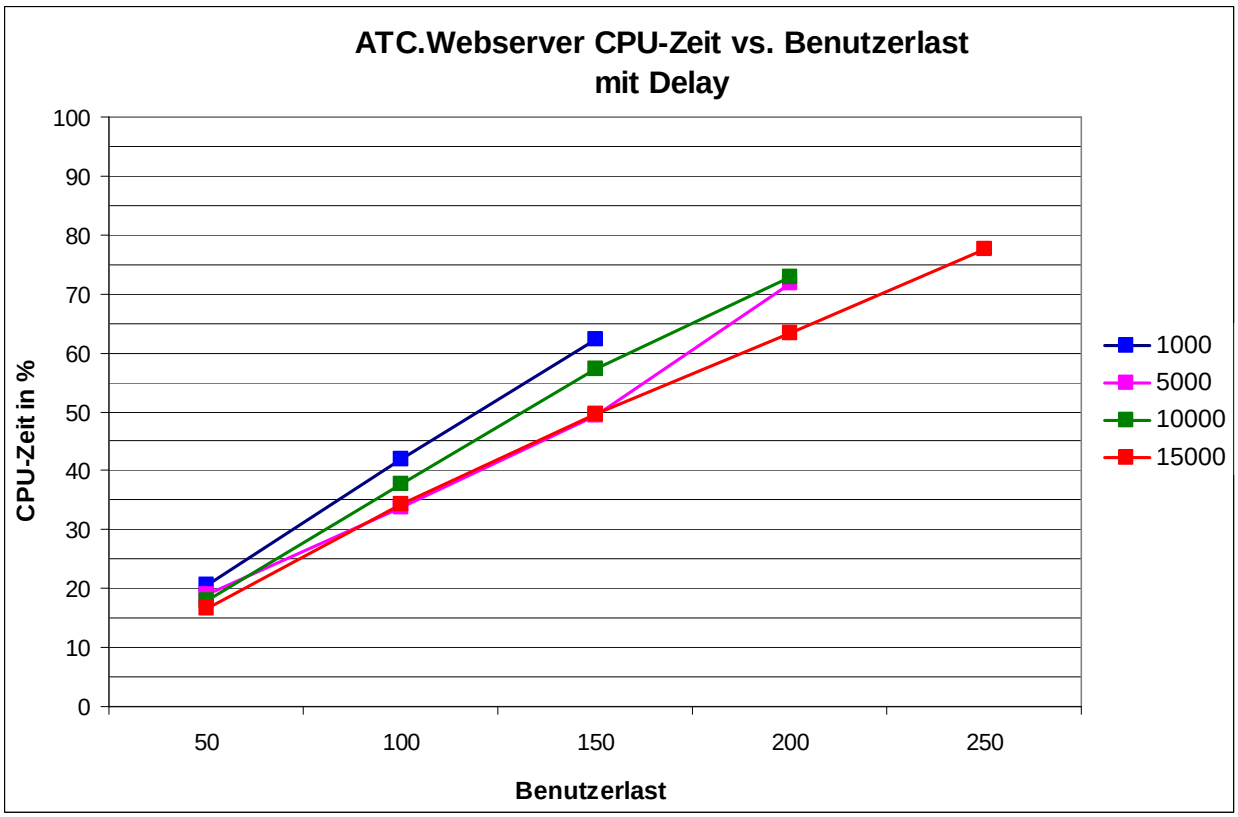


Abb. 4.4.2.21: ATC.Webserver CPU-Zeit vs. Benutzerlast mit Delay.

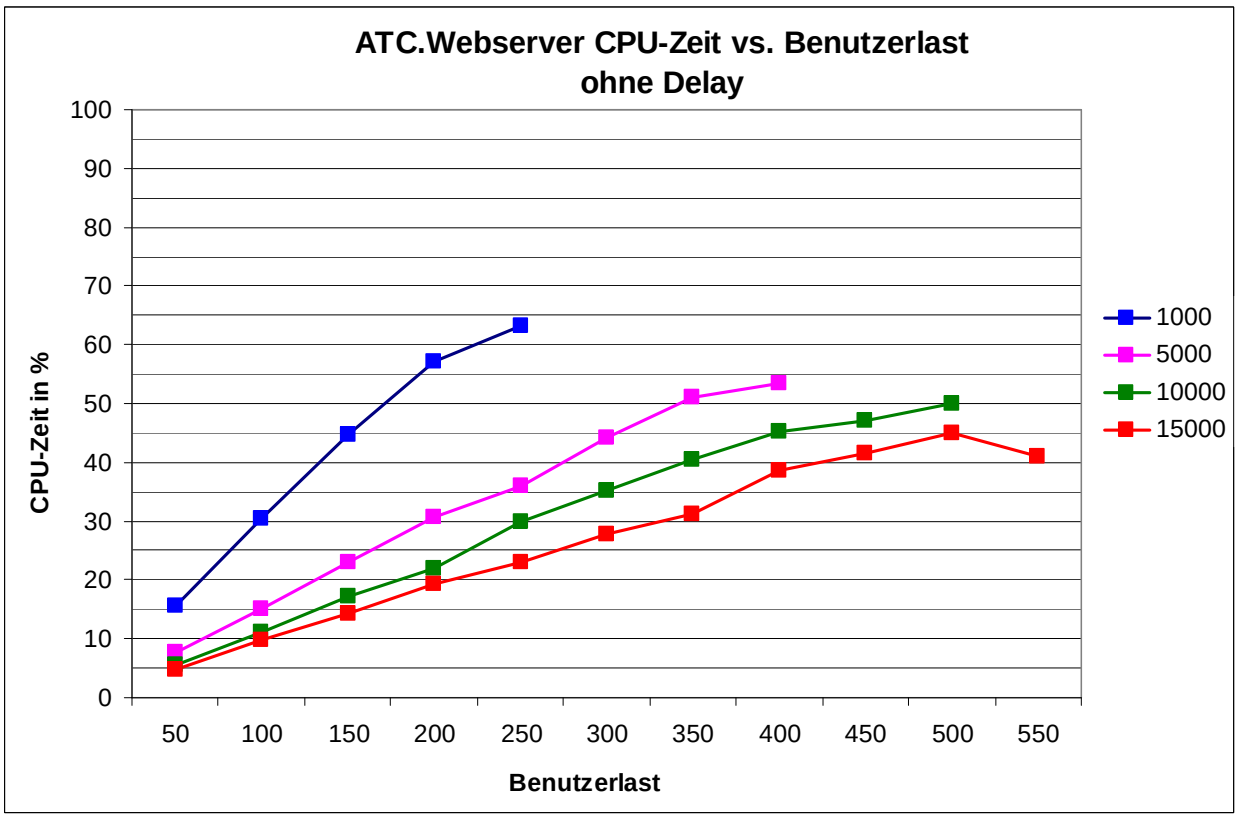


Abb. 4.4.2.22: ATC.Webserver CPU-Zeit vs. Benutzerlast ohne Delay.

Vergleich ATC.Webserver CPU-Zeit in Prozent					
Update-Rate	**Delay**	**Benutzerlast**			
		50	**100**	**150**	**200**
1000	mit	20,63	42,03		
	ohne	15,51	30,52		
Differenz		5,12	11,51		
5000	mit	19,10	33,83	49,36	
	ohne	7,73	15,07	23,15	
Differenz		11,37	18,76	26,22	
10000	mit	17,93	37,82	57,34	
	ohne	5,67	11,20	17,10	
Differenz		12,26	26,62	40,24	
15000	mit	16,52	34,42	49,65	63,35
	ohne	4,77	9,83	14,30	19,30
Differenz		11,75	24,58	35,35	44,06

Tabelle 4.4.2.4: Vergleich der Messwerte zur CPU-Zeit der ATC.Webservers vs. Benutzerlast.

- Leistungsmetrik 9: *ATC.Webserver* CPU-Zeit vs. Benutzerlast

In den dargestellten Abbildungen 4.4.2.21 und 4.4.2.22 wird das Verhalten der verwendeten CPU-Zeit des *ATC.Webservers* mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.2.21 wird die verwendete CPU-Zeit *mit Delay* betrachtet und in Abbildung 4.4.2.22 *ohne Delay*. Zudem werden in beiden Abbildungen die unterschiedlichen Update-Raten berücksichtig. In der Tabelle 4.4.2.4 werden zusätzlich die verschiedenen Messreihen gegenübergestellt.

In den beiden Abbildungen 4.4.2.21 und 4.4.2.22 ist bis zum Erreichen der maximalen Benutzerlast ebenfalls ein eher proportionales Verhalten der benötigten CPU-Zeiten zur Benutzerlast zu erkennen. Zu mindestens sind keine erkennbar langen CPU-Zeiten vorhanden. Der *ATC.Webserver* ist somit ebenfalls skalierbar.

Besonders auffallend ist auch hier wieder der Unterschied der benötigten CPU-Zeiten in den beiden Delay-Einstellungen in Abhängigkeit zur Benutzerlast und zur Update-Rate. Dieses unterschiedliche Verhalten wird in Tabelle 4.4.2.4 sehr deutlich dargestellt. Die Unterschiede sind im Vergleich zu den CPU-Zeiten des *ATC.Servers* (siehe Leistungsmetrik 8) jedoch durchgehend bei allen Update-Raten vorhanden. Je größer die Update-Rate (> 1000 ms) und die Benutzerlast ist, desto größer fallen die Unterschiede in den benötigten CPU-Zeiten zwischen den beiden Delay-Einstellungen aus. Zum Teil entspricht die benötigte CPU-Zeit *mit Delay* dem Dreifachen der CPU-Zeit *ohne Delay*.

Das Verhalten der CPU-Zeiten des *ATC.Webservers* steht aufgrund der Kommunikation mit dem *ATC.Server* zwar in einer gewissen Abhängigkeit zu diesem, allerdings sind die Unterschiede der CPU-Zeiten des *ATC.Webservers* darauf nicht zurückzuführen. Denn im Vergleich zum *ATC.Server* sind die Unterschiede der CPU-Zeiten zwischen den beiden Delay-Einstellungen in allen Update-Raten vorhanden. Wahrscheinlich sind die unterschiedlichen CPU-Zeiten auf die bereits beschriebene Problematik mit der Serialisierung der Flugzeugnachrichten in das JSON-Format zurückzuführen (siehe Leistungsmetrik 6). Das würde auch das Ansteigen der Unterschiede in den CPU-Zeiten mit einer steigenden Benutzerlast und Update-Rate erklären.

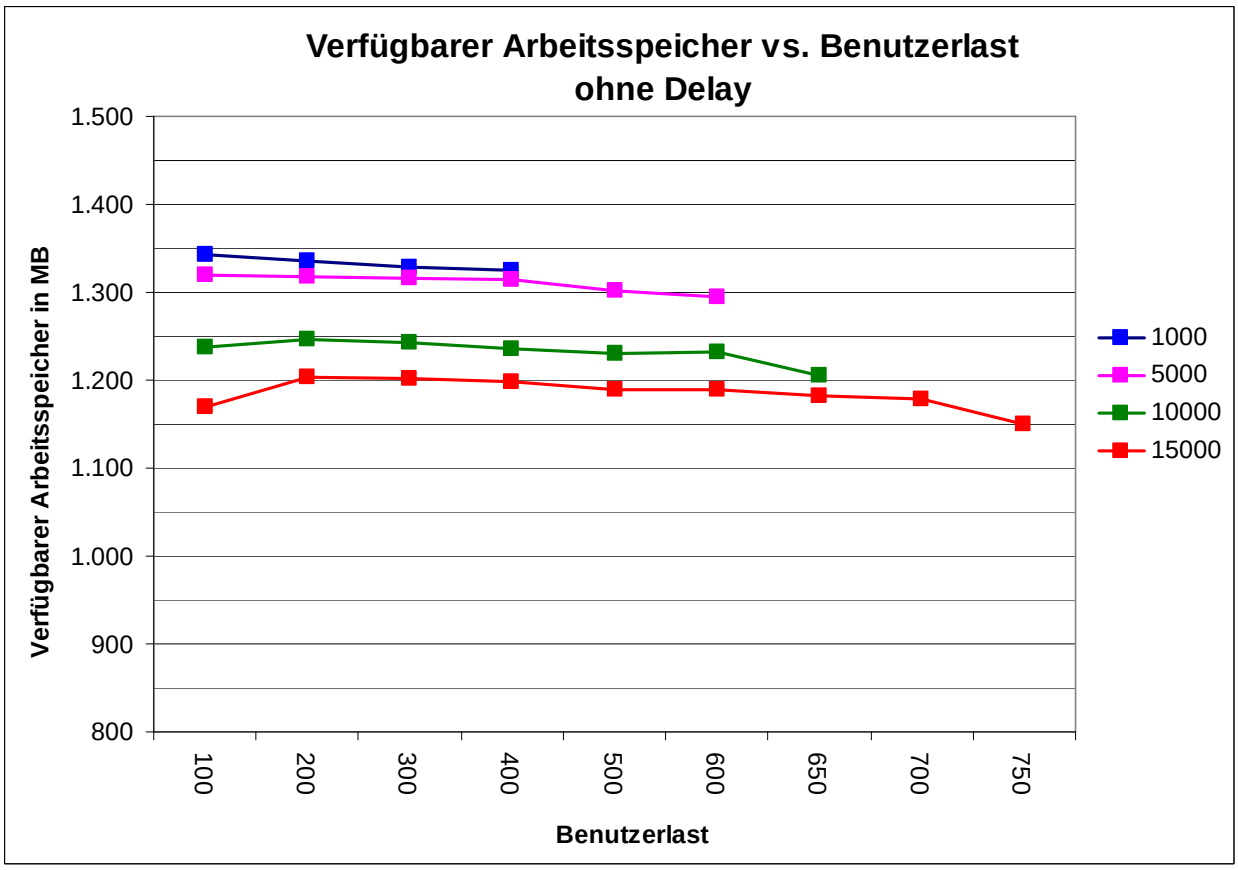


Abb. 4.4.1.23 Verfügbarer Arbeitsspeicher vs. Benutzerlast ohne Delay.

- Leistungsmetrik 10: Verfügbarer Arbeitsspeicher vs. Benutzerlast

In der Abbildung 4.4.1.23 wird das Verhalten des verfügbaren Arbeitsspeichers mit einer steigenden Benutzerlast ohne Delay dargestellt. Im Verhalten zu den beiden Delay-Einstellungen (mit und ohne) bestehen keinerlei Unterschiede, so dass stellvertretend der *Verfügbare Arbeitsspeicher* ohne Delay betrachtet wird. In der Abbildung werden die unterschiedlichen Update-Raten berücksichtig.

Nach der Abbildung 4.4.1.23 nimmt der verfügbare Arbeitsspeicher mit einer steigenden Benutzerlast zwar etwas ab, allerdings sind sonst keinerlei Auffälligkeiten zu erkennen. Die verschiedenen Ausgangslagen des verfügbaren Arbeitsspeichers resultieren vermutlich aus den gesammelten Flugzeugnachrichten die der Flugdatenserver in einem Ringbuffer speichert.

4.4.3 Testszenario Teilsystem

In diesem Abschnitt werden die Ergebnisse aus der Leistungsanalyse des *Teilsystems* dargestellt und bewertet. Die Beschreibung erfolgt nicht so ausführlich wie zuvor bei der Leistungsanalyse des *Gesamtsystems*. Es wird hauptsächlich auf Unterschiede im Leistungsverhalten zum *Gesamtsystem* eingegangen.

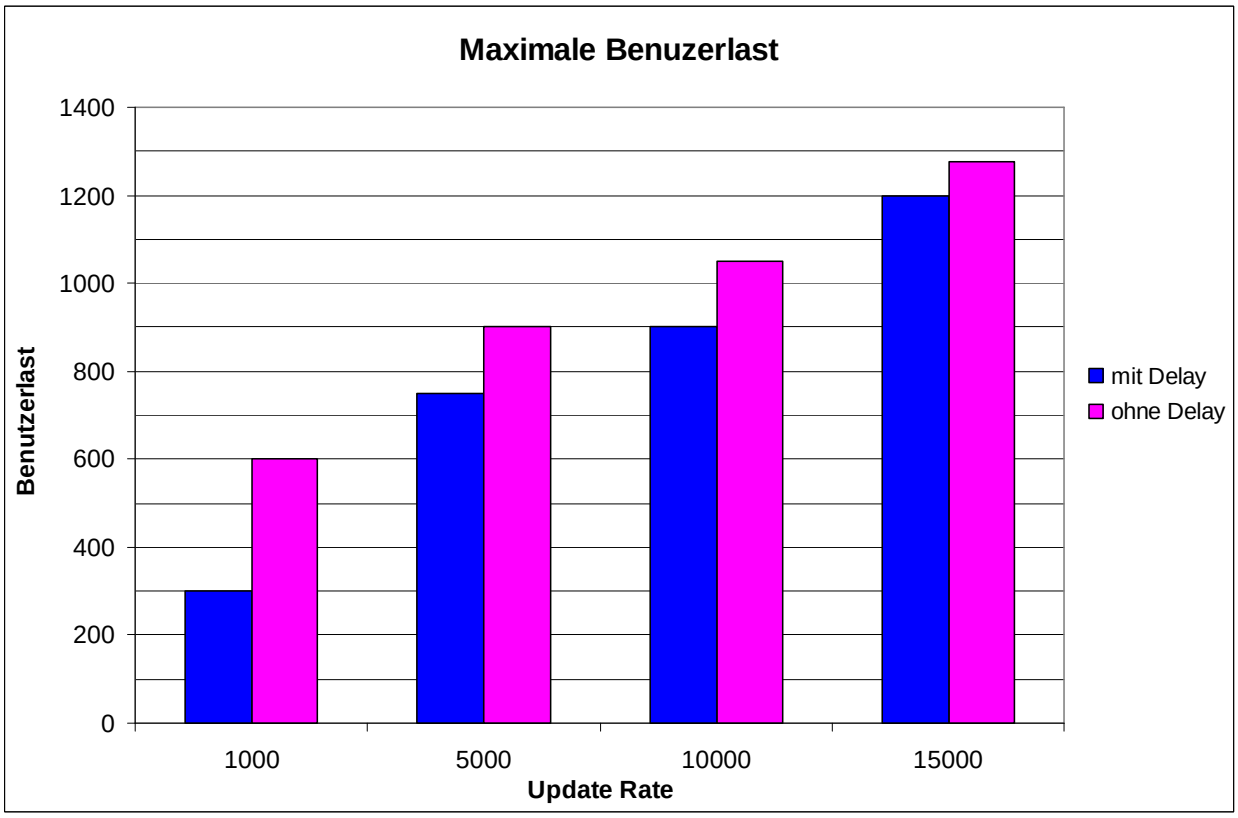


Abb. 4.4.3.1 Maximale Benutzerlasten bei der Analyse des Teilsystems

Mit der Durchführung der in Tabelle 4.1.1 definierten Testläufe konnten die in Abbildung 4.4.3.1 dargestellten maximalen Benutzerlasten (*Peak Loads*), die der Flugdatenserver verarbeiten kann, ermittelt werden. Im Vergleich zur Analyse des *Gesamtsystems,* können mit der Analyse des *Teilsystems* weitaus größere Benutzerlasten verarbeitet werden, da nur der *ATC.Server-Prozess* ausgeführt wird und die CPU beansprucht (beim Testszenario *Gesamtsystem* wurden grundsätzlich immer beide Flugdatenserver-Prozesse ausgeführt).

Wie in der Abbildung 4.4.3.1 dargestellt, kann der Flugdatenserver auch hier wieder unterschiedliche maximale Benutzerlasten, in Abhängigkeit zur Update-Rate und zur Delay-

Einstellung, verarbeiten. Es besteht somit ein ähnliches Verhalten wie zuvor beim Testszenario *Gesamtsystem*. Allerdings verhalten sich die unterschiedlichen maximalen Benutzerlasten zwischen den beiden Delay-Einstellung verschieden. Sind die Unterschiede der maximalen Benutzerlasten in den beiden Delay-Einstellungen beim Gesamtsystem mit höheren Update-Raten steigend, so werden diese beim Teilsystem mit höheren Update-Raten eher geringer. Dieser mengenmäßige Unterschied der maximalen Benutzerlasten zwischen den beiden Delay-Einstellungen wird in der Tabelle 4.4.3.1 noch einmal deutlich.

Mengenmäßiger Unterschied der maximalen Benutzerlasten zwischen den beiden Delay-Einstellungen				
Testszenario	**Update-Rate**			
	1000	**5000**	**10000**	**15000**
Gesamtsystem	100	200	300	300
Teilsystem	300	150	150	75

Tabelle. 4.4.3.1 Mengenmäßige Unterschiede der maximalen Benutzerlasten in den Delay-Einstellungen.

Wie in Abschnitt 4.4.2 beschrieben resultieren die gravierenden Unterschied beim Testszenario *Gesamtsystem* zwischen den beiden Delay-Einstellungen vor allem aufgrund der unterschiedlichen Mengen an bereitgestellten Flugzeugnachrichten und der damit zusammenhängenden Problematik mit der JSON-Serialisierung.

Inwiefern die unterschiedlichen maximalen Benutzerlasten beim Testszenario *Teilsystem* zustande kommen, kann leider nicht exakt bestimmt werden. Es können mit Hilfe der Messreihen der verschiedenen Leistungsmetriken nur Hinweise gegeben werden. Nachfolgend werden einige Messwerte verschiedener Leistungsmetriken zur Analyse des *Teilsystems* beschrieben.

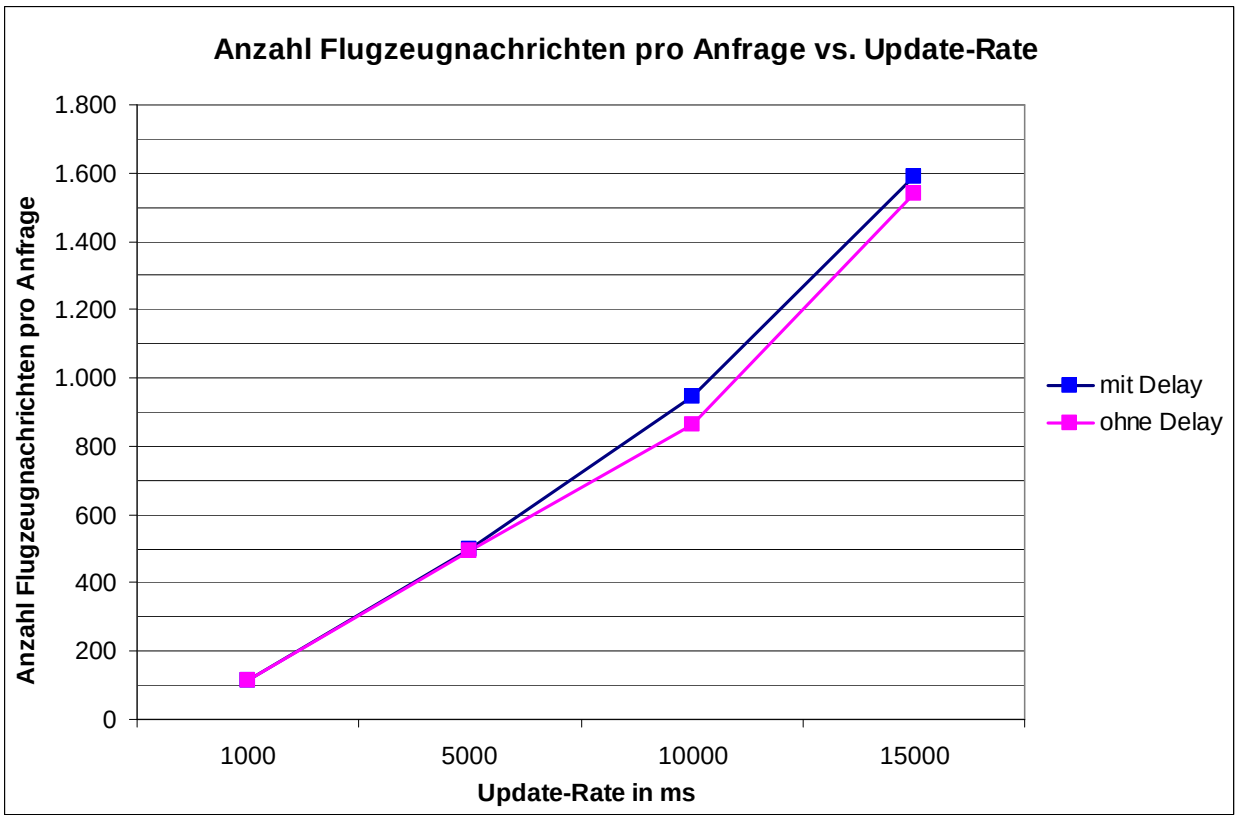


Abb. 4.4.3.2 Anzahl Flugzeugnachrichten pro Anfrage vs. Update-Rate

- Leistungsmetrik 1: Anzahl Flugzeugnachrichten pro Anfrage vs. Update-Rate

In der Abbildung 4.4.3.2 wird das Verhalten der Anzahl Flugzeugnachrichten pro *Anfrage* zur Update-Rate, unter Betrachtung der beiden Delay-Einstellungen, dargestellt.

Im Vergleich zu dem Ergebnis aus der Analyse des *Gesamtsystems,* bestehen zwischen den beiden Delay-Einstellungen keine Unterschiede. Der *ATC.Server* stellt dem *ATC.Webserver* in Abhängigkeit zur verwendeten Update-Rate eine bestimmte Menge an Flugzeugnachrichten bereit. Grundsätzlich wird also bei jeder Anfrage vom *ATC.Webserver* zum *ATC.Server* die in der Abbildung dargestellte Menge an Flugzeugnachrichten in Abhängigkeit zur Update-Rate übertragen. Die Abhängigkeit zur Update-Rate wurde bereit bei der Darstellung der Ergebnisse zum Gesamtsystem beschrieben (siehe Abschnitt 4.4.2 *Leistungsmetrik 4*).

Der Optimierungsmechanismus, der in Kapitel 2.2.4 beschrieben ist, macht sich auf dem *ATC.Server* also nicht bemerkbar, da dieser im *Privilegierten-Web-Service* auf dem *ATC.Webserver* implementiert ist.

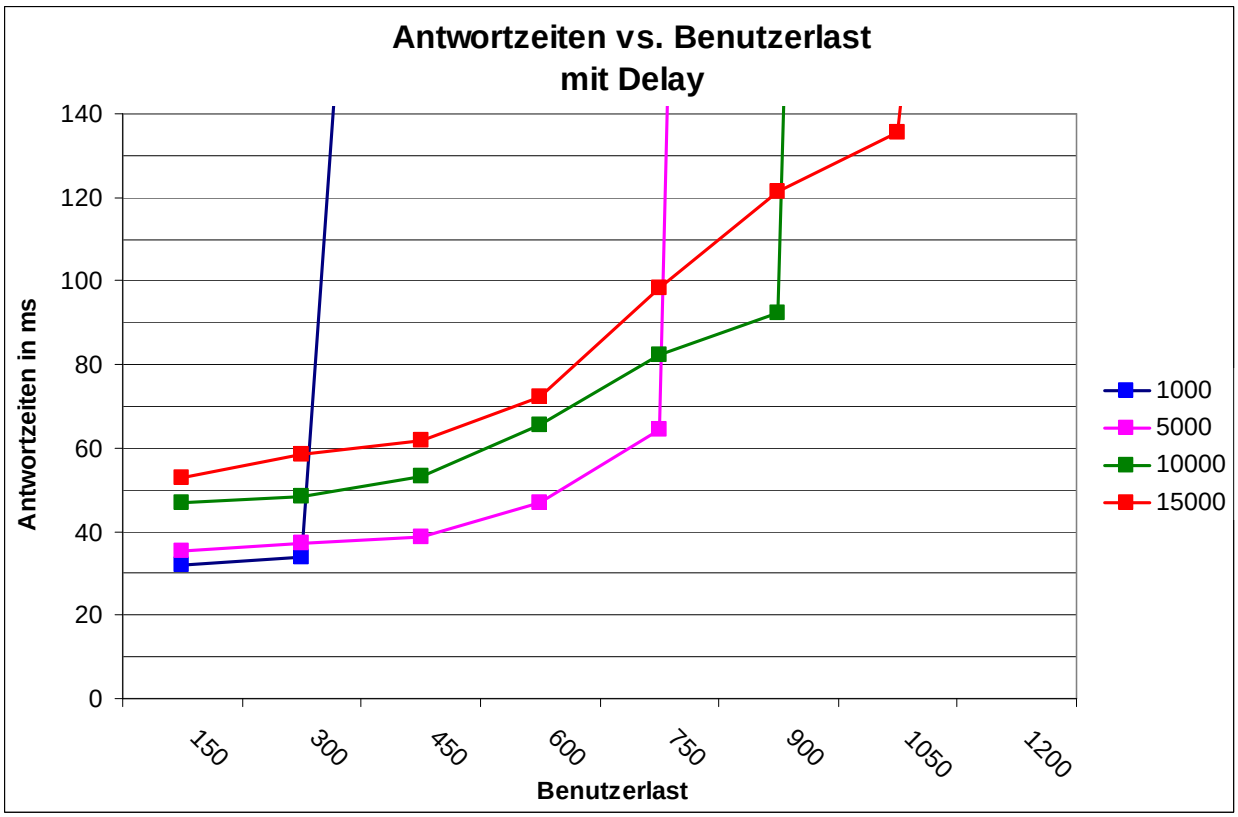


Abb. 4.4.3.3: Antwortzeiten vs. Benutzerlast mit Delay.

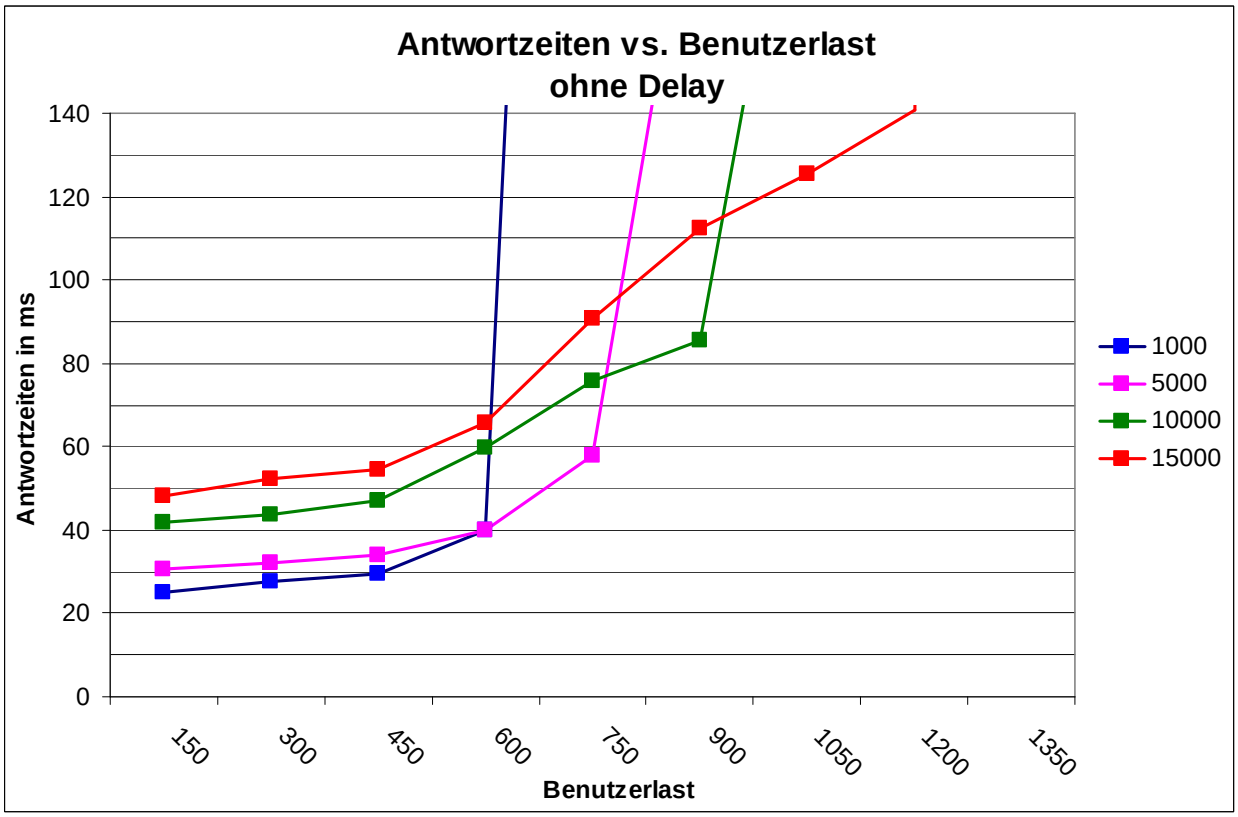


Abb. 4.4.3.4: Antwortzeiten vs. Benutzerlast ohne Delay.

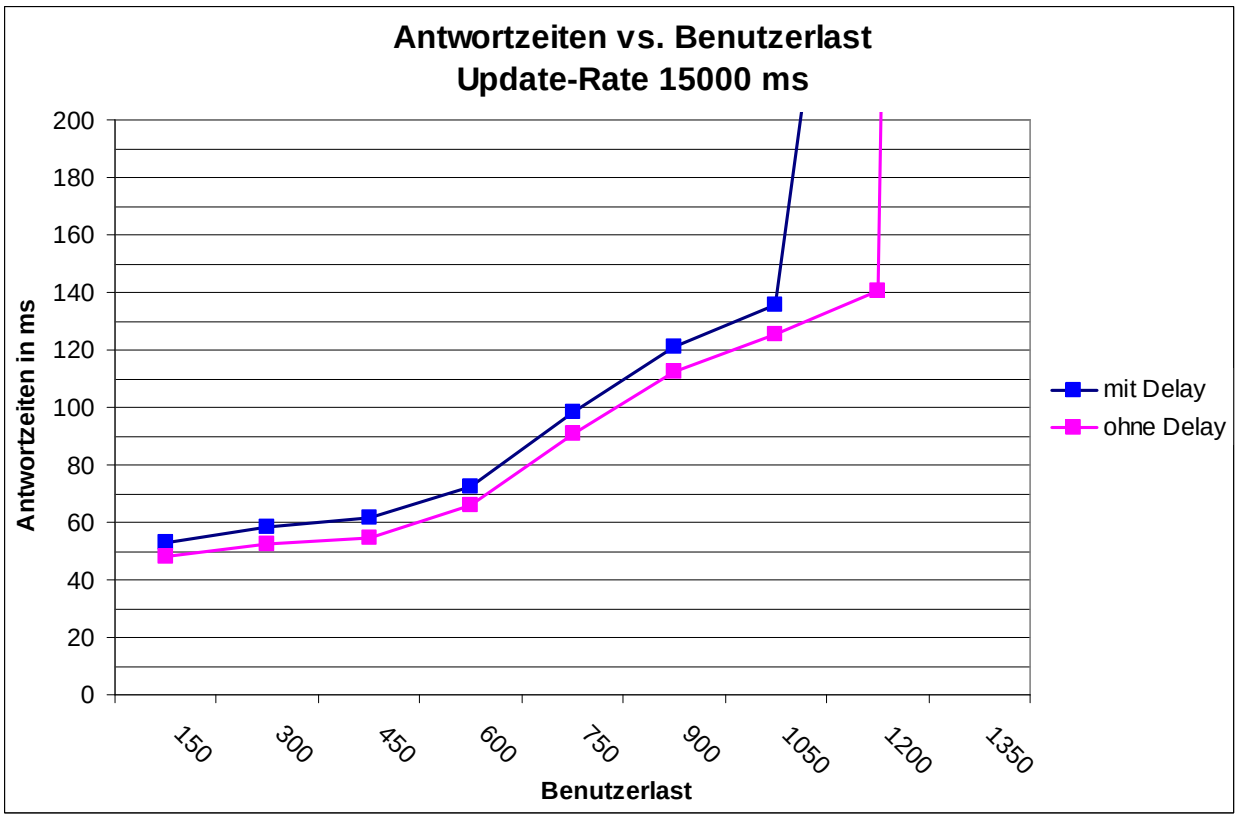


Abb. 4.4.3.5: Antwortzeiten vs. Benutzerlast mit und ohne Delay.

- Leistungsmetrik 2: Antwortzeiten vs. Benutzerlast

In den Abbildungen 4.4.3.3 und 4.4.3.4 wird das Verhalten der Antwortzeiten auf die Client-Anfragen mit einer steigenden Benutzerlast beschrieben. In Abbildung 4.4.3.3 werden die Antwortzeiten *mit Delay* und in Abbildung 4.4.3.4 *ohne Delay* betrachtet. In beiden Abbildungen werden zudem die unterschiedlichen Update-Raten berücksichtigt. In der Abbildung 4.4.3.5, wird das unterschiedliche Verhalten der Antwortzeiten zur Delay-Einstellung am Beispiel einer Updaterate von 15000 ms dargestellt.

Die hier gemessene Antwortzeit enthält die Verarbeitungszeit des RPC im *ATC.Server* und die Übertragungszeit vom *ATC.Server* zu einem simulierten Benutzer. Die Übertragungszeiten in beiden Delay-Einstellungen sollten in etwa gleich ausfallen, da die Menge der Flugzeugnachrichten, die zurückgegeben werden, in beiden Delay-Einstellungen gleich ausfällt. Somit kann über die Antwortzeit eventuell Rückschlüsse auf die Verarbeitungszeit eines RPC im *ATC.Server* gezogen werden. Ansonsten besteht keine Möglichkeit die Verarbeitungszeit eines RPC zu messen. Nur anhand der Verarbeitungszeiten kann definitiv nachgewiesen

werden, ob die unterschiedliche Datenbereitstellung für die Berücksichtigung des Delays für die Leistungsunterschiede mit den beiden Delay-Einstellungen verantwortlich ist.

Aus den beiden Abbildungen 4.4.3.3 und 4.4.3.4 ist zu erkennen, dass sich die Antwortzeiten ähnlich wie beim Test des *Gesamtsystems* verhalten. Bis zu einer bestimmten Benutzerlast steigen die Antwortzeiten nur leicht an, danach allerdings deutlich stärker. Mit dem Überschreiten der maximalen Benutzerlast, steigen die Antwortzeiten dann sehr stark. Wie bereits in Abschnitt 4.4.2 erklärt, resultiert dieses Verhalten aus der steigenden Verarbeitungszeit auf dem Flugdatenserver mit einer steigenden Benutzerlast.

Auffallend sind zudem die Unterschiede in den Antwortzeiten zwischen den beiden Delay-Einstellungen. Der Unterschied wird vor allem in Abbildung 4.4.3.5 deutlich. So liegen die Antwortzeiten *mit Delay* mit einer steigenden Benutzerlast immer etwas oberhalb der Antwortzeiten *ohne Delay*. Dies könnte ein Hinweis darauf sein, dass die Verarbeitungszeiten der Datenbereitstellung für die beiden Delay-Einstellungen auf dem *ATC.Server* tatsächlich unterschiedlich sind und deshalb unterschiedliche maximale Benutzerlasten verarbeitet werden können.

Die Antwortzeiten liefern somit einen Hinweis darauf, dass die unterschiedliche Datenbereitstellung für die beiden Delay-Einstellungen (mit oder ohne) dafür verantwortlich ist, dass unterschiedliche maximale Benutzerlasten verarbeitet werden können.

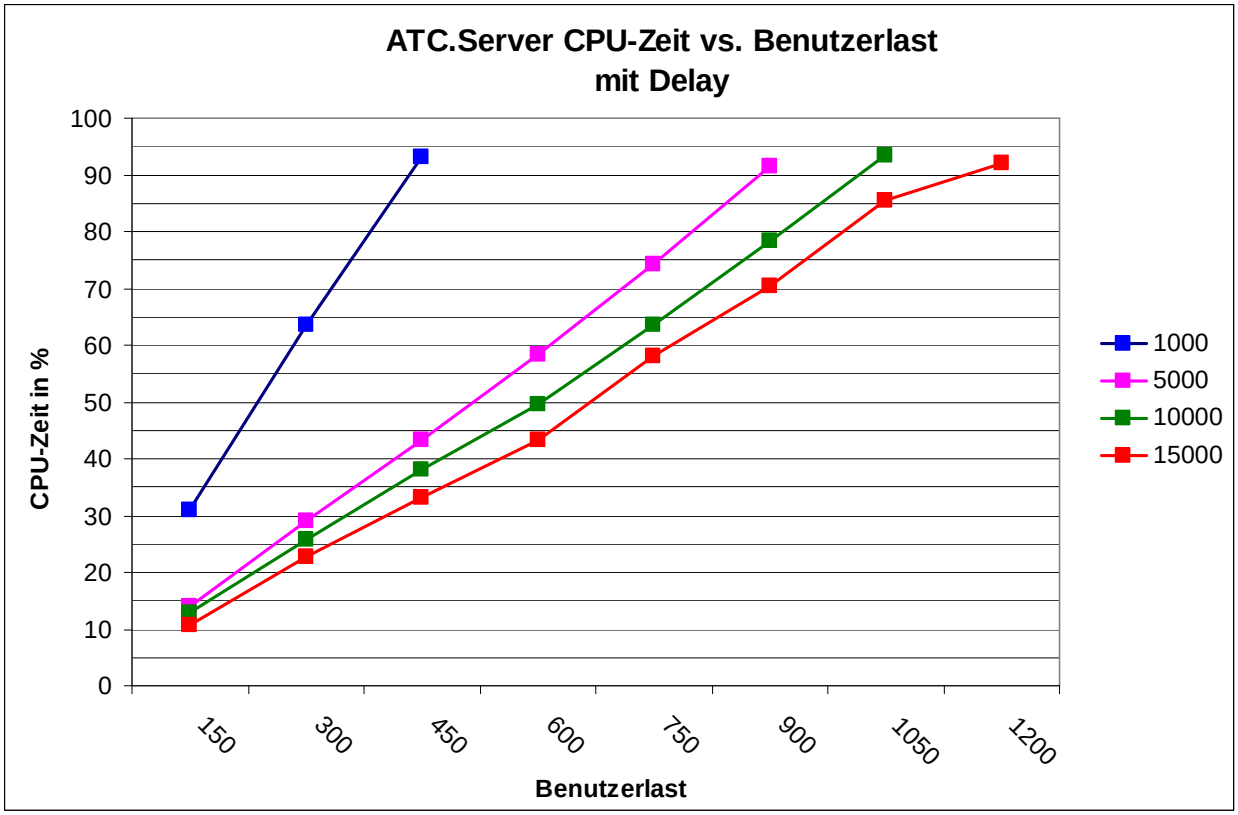


Abb. 4.4.3.6: ATC.Server CPU-Zeit vs. Benutzerlast mit Delay.

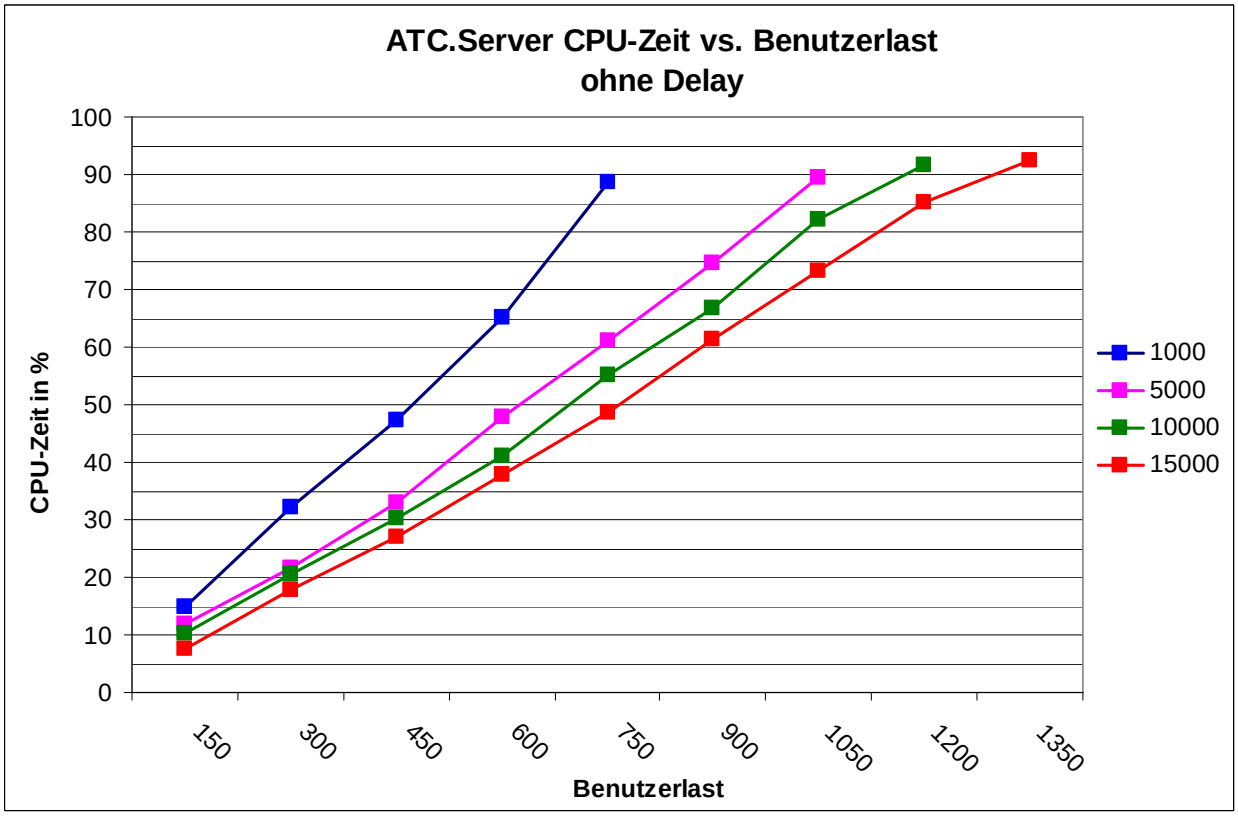


Abb. 4.4.3.7: ATC.Server CPU-Zeit vs. Benutzerlast ohne Delay.

Vergleich ATC.Server CPU-Zeit in Prozent									
Update-Rate	**Delay**	**Benutzerlast**							
		150	**300**	**450**	**600**	**750**	**900**	**1050**	**1200**
1000	mit	30,88	63,60						
	ohne	14,87	32,09						
Differenz		16,02	31,51						
5000	mit	14,10	29,01	43,22	58,36	74,30			
	ohne	11,76	21,54	33,09	47,97	61,14			
Differenz		2,33	7,47	10,12	10,39	13,16			
10000	mit	12,97	25,75	38,15	49,55	63,69	78,41		
	ohne	10,30	20,49	30,21	41,10	55,04	66,75		
Differenz		2,67	5,26	7,94	8,45	8,65	11,66		
15000	mit	10,73	22,71	33,26	43,16	57,99	70,52	85,45	92,13
	ohne	7,62	17,79	27,02	37,89	48,52	61,41	73,30	85,21
Differenz		3,11	4,92	6,24	5,27	9,48	9,11	12,16	6,92

Tabelle. 4.4.3.2 Vergleich der Messwerte zur CPU-Zeit des ATC.Servers vs. Benutzerlast

- Leistungsmetrik 3: ATC.Server CPU-Zeit vs. Benutzerlast

In den dargestellten Abbildungen 4.4.3.6 und 4.4.3.7 wird das Verhalten der verwendeten CPU-Zeit des *ATC.Servers* mit einer steigenden Benutzerlast dargestellt. In Abbildung 4.4.3.6 wird die verwendete CPU-Zeit *mit Delay* betrachtet und in Abbildung 4.4.3.7 *ohne Delay*. Zudem werden in beiden Abbildungen die unterschiedlichen Update-Raten berücksichtig. In der Tabelle 4.4.3.2 werden zusätzlich die verschiedenen Messreihen gegenübergestellt.

Wie in der Analyse des Gesamtsystems, wird in den Abbildungen 4.4.3.6 und 4.4.3.7 deutlich, dass die verwendete CPU-Zeit auch hier wieder eher proportional mit der Benutzerlast ansteigt.

Auffallend sind allerdings die Unterschiede in den verwendeten CPU-Zeiten zwischen den beiden Delay-Einstellungen bei gleicher Update-Rate, siehe Tabelle 4.4.3.2. Besonders ausgeprägt ist dieser Unterschied bei einer Update-Rate von 1000 ms. Hier entspricht die verwendete CPU-Zeit *mit Delay* der doppelten CPU-Zeit *ohne Delay*. Bei größeren Update-Raten (> 1000 ms) bestehen die Unterschiede in der verwendeten CPU-Zeit ebenfalls. Hier sind die Differenzen der CPU-Zeiten bei geringer Benutzerlast noch relativ gering, steigen aber mit einer steigenden Benutzerlast. Die Differenzen die sich ergeben, scheinen von einer hohen Anzahl an Client-Anfragen pro Sekunden abhängig zu sein.

Mit der Beschreibung der Ergebnisse zur Leistungsmetrik 2 und 3 liegt die Vermutung nahe, dass die unterschiedliche Implementierung der Datenbereitstellung dafür verantwortlich ist, dass der Flugdatenserver ohne Berücksichtigung eines Delays größere Benutzerlasten verarbeiten kann. Wie schon einleitend in diesem Abschnitt erwähnt, bedarf es einer tiefer gehenden Analyse, z.B. mit einem Profiling Tool, um die Ursachen für die Unterschiede in der Leistung der beiden Delay-Einstellungen festzustellen.

Allerdings ist es sehr wahrscheinlich, dass die Unterschiede in der Leistung aufgrund der unterschiedlichen Implementierung in der Datenbereitstellung verursacht werden. Die unterschiedliche Implementierung wurde mit Pseudocode in Abschnitt 2.2.4 ausführlich beschrieben.

4.4.4 Testszenario Gesamtsystem als Hüllentest

Mit dem Hüllentest soll ermittelt werden, inwiefern die Kommunikation zwischen den beiden Komponenten des Flugdatenservers (*ATC.Server* und *ATC.Webserver*) über den *TcpChannel* die Leistung beeinflusst. Hierfür wird der *ATC.Server* durch den entwickelten Dummy ersetzt. Dieser Dummy hat lediglich die Aufgabe, den *ATC.Webserver* mit einer konfigurierbaren Menge an Flugzeugnachrichten zu versorgen. Die restliche Funktionalität des *ATC.Servers*, welche die CPU zusätzlich beanspruchen könnte, ist im Dummy nicht vorhanden.

Wie in Abschnitt 4.3 in Tabelle 4.3.1 aufgeführt, wurde der *Hüllentest* nur mit einer einzigen Konfiguration (Update-Rate = 1000 ms, mit Delay) zu einer steigenden Benutzerlast ausgeführt. Dabei hat sich herausgestellt, dass mit der Ausführung des Hüllentests nur unwesentlich mehr Leistung erreicht wird, als bei der Analyse des *Gesamtsystems*. Konnten mit der getesteten Konfiguration (Update-Rate = 1000 ms, mit Delay) bei der Analyse des *Gesamtsystems* maximal 100 Benutzer verarbeitet werden, so sind es beim Hüllentest gerade einmal 150 Benutzer. Nachfolgend wird das Ergebnis des *Hüllentests* anhand der Betrachtung der CPU-Auslastung und der CPU-Zeiten der beiden Flugdatenserver-Prozesse *ATC.Server* und *ATC.Webserver* analysiert.

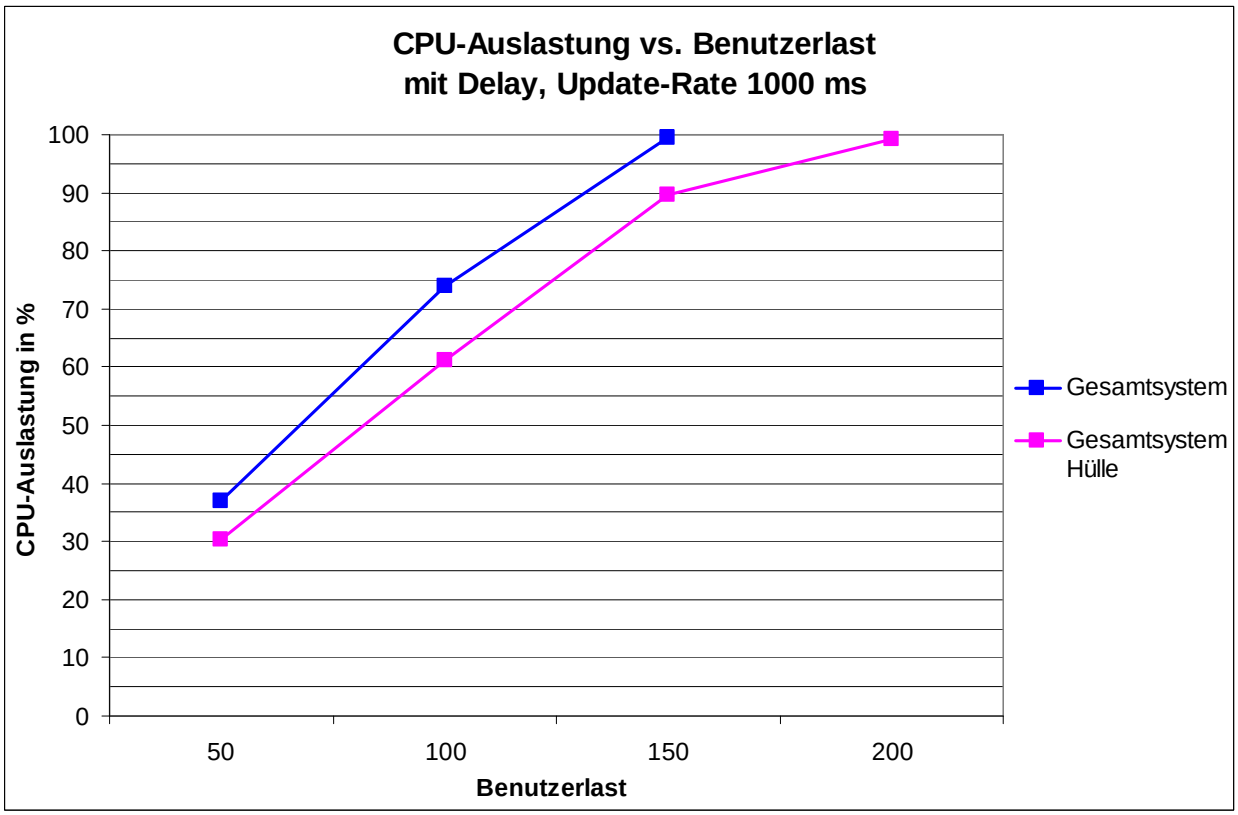


Abb. 4.4.4.1: CPU-Auslastung vs. Benutzerlast.

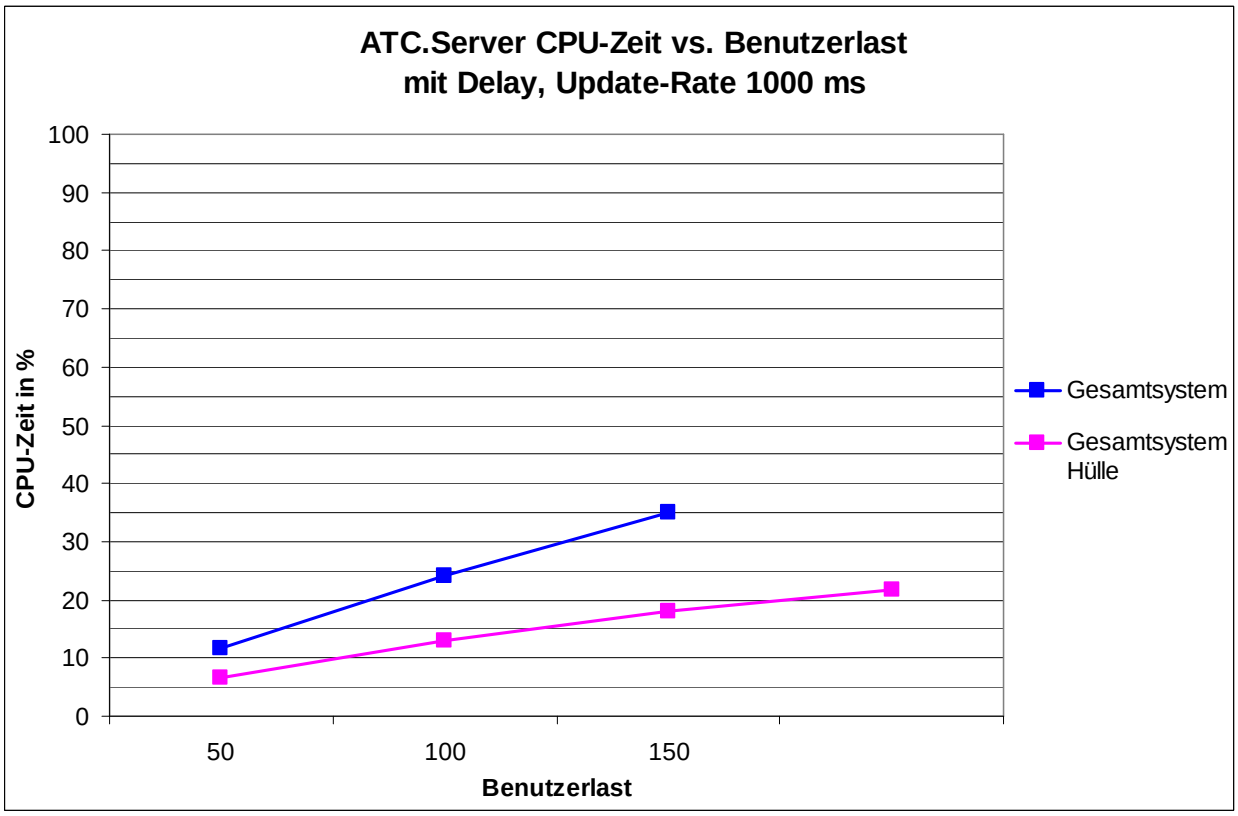


Abb. 4.4.4.2: ATC.Server CPU-Zeit vs. Benutzerlast.

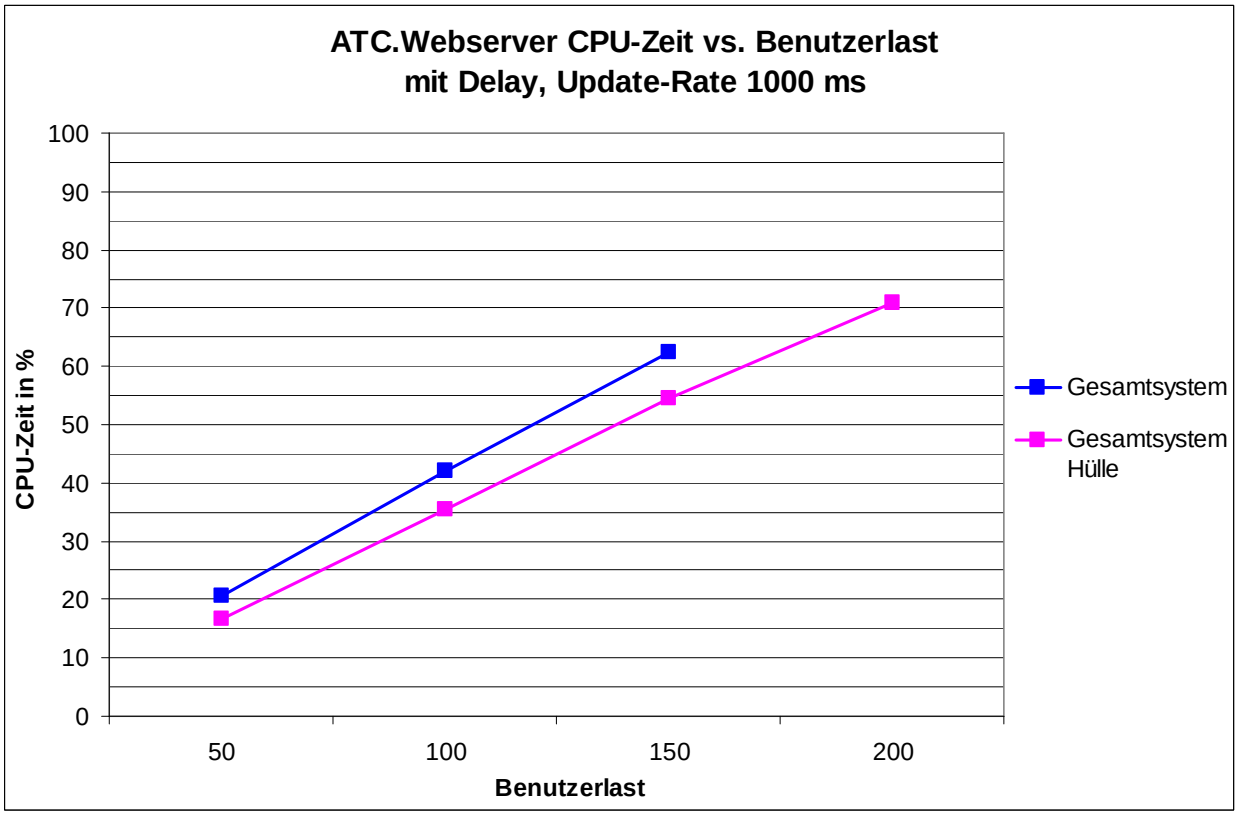


Abb. 4.4.4.3: ATC.Webserver CPU-Zeit vs. Benutzerlast.

In den Abbildungen 4.4.4.1, 4.4.4.2 und 4.4.4.3 werden die CPU-Auslastung des Flugdatenservers und die CPU-Zeiten der beiden Flugdatenserver-Prozesse mit einer steigenden Benutzerlast dargestellt. Dabei erfolgt jeweils ein Vergleich der CPU-Auslastung der beiden Testszenarien *Gesamtsystem* und *Gesamtsystem als Hüllentest*.

Wie bei den Ergebnissen der beiden Testszenarien *Gesamtsystem* und *Teilsystem* verläuft die CPU-Auslastung ebenfalls proportional mit einer steigenden Benutzerlast. Bei den CPU-Zeiten der beiden Flugdatenserver-Prozesse ist dies ebenfalls der Fall.

In den Abbildungen wird deutlich, inwiefern sich die weitaus geringere Funktionalität des *Dummys* bemerkbar macht. So sind die CPU-Zeiten der beiden Flugdatenserver-Prozesse beim *Hüllentest* etwas geringer als bei der Analyse des *Gesamtsystems*. Dadurch kann mit dem Hüllentest auch eine etwas größere Benutzerlast verarbeitet werden.

Im Grunde genommen wird mit dem Hüllentest aber noch einmal deutlich, dass trotz geringster Funktionalität im Dummy (Rolle des *ATC.Servers*), also nur die Weitergabe einer bestimmten Menge an Flugzeugnachrichten, die Leistung des Flugdatenservers maßgeblich durch die Kommunikation der beiden Flugdatenserver-Prozesse beeinflusst wird. Gerade durch den Umstand, dass die beiden Prozesse auf demselben System ausgeführt werden, wird die CPU sehr stark beansprucht.

4.4.5 Fazit

Mit der Leistungsanalyse der drei Testszenarien konnten umfassende Kenntnisse und Abhängigkeiten zwischen Leistungsindikatoren ermittelt werden, die das Leistungsverhalten des Flugdatenservers maßgeblich bestimmen. Grundsätzlich hat sich das Leistungsverhalten des Flugdatenservers als skalierbar gezeigt. Eine doppelte Benutzerlast konnte mit einer doppelten Ressourcenverwendung kompensiert werden. Auf Basis der Erkenntnisse der Leistungsanalyse konnten zudem einige Leistungsengpässe ermittelt werden. Nachfolgend werden diese Leistungsengpässe noch einmal zusammengefasst.

- Ausführung der Flugdatenserver-Prozesse auf demselben System
  Die Ausführung der beiden Flugdatenserver-Prozesse *ATC.Server* und *ATC.Webserver* auf demselben System (Hardware), bremst die Leistung des Flugdatenservers. Dadurch, dass die beiden Prozesse auf demselben System ausgeführt werden, nehmen beide Prozesse die vorhandenen Systemressourcen (CPU) in Anspruch. Dies führt dazu, dass die Systemressourcen dementsprechend schneller ausgelastet sind. Würden die Prozesse auf unterschiedlichen Systemen ausgeführt werden, hätte jeder Prozess seine eigenen Ressourcen zur Verfügung.

  Zudem erzeugen die beiden Prozesse unnötigen Leistungs-Overhead durch die Ausführung auf demselben System. Wie aus der Analyse des Gesamtsystems aus Abschnitt 4.4.2 hervorging, wird für die Verarbeitung einer Client-Anfrage in den beiden Prozessen jeweils ein Thread verwendet. Dies führt bei einer großen Benutzerlast und einer geringen Update-Rate zu entsprechend vielen aktiven Threads. Dadurch werden wiederum enorm viele Kontextwechsel pro Sekunde erzeugt, was wiederum zu verlängerten Wartenzeiten in der Verarbeitung der einzelnen Threads führt. Würden die beiden Prozesse auf unterschiedlichen Systemen ausgeführt werden, würden sich diese Kontextwechsel auf die beide Systeme aufteilen. Wie bereits in Abschnitt 4.4.2 beschrieben, beanspruchen eine hohe Anzahl an Kontextwechseln pro Zeiteinheit die CPU entsprechend (siehe Abschnitt 4.4.2, Leistungsmetrik 3).

- Menge der Flugzeugnachrichten / JSON-Serialisierung
  Ein weiterer Leitungsengpass entsteht durch die Menge der Flugzeugnachrichten, die einer Client-Anfrage zurückgegeben werden. Wie in Abschnitt 4.4.2 zur *Leistungsmetrik 6* beschrieben, führen eine große Menge an Flugzeugnachrichten insgesamt zu einer längeren Ausführungszeit der Web-Service-Anfragen auf dem *ATC.Webserver*. Diese

längere Ausführungzeit resultiert aus der Serialisierung der Flugzeugnachrichten in das JSON-Format für die Web-Service-Antwort. Hierbei hat sich herausgestellt, dass die Serialisierung in das JSON-Format 20% bis 25% langsamer ist, als in das XML-Format. Dieser Umstand hat sich dementsprechend in den Ausführungs- und Antwortzeiten der Web-Service-Anfragen bemerkbar gemacht.

Zudem beeinflusst die Menge der Flugzeugnachrichten auch entsprechend das Datenvolumen das über das Netzwerk versendet wird. Zum einen vom *ATC.Server* zum *ATC.Webserver* und zum anderen vom *ATC.Webserver* zum Web-Client. Je nach Datenvolumen, fällt hier eine entsprechende Übertragungszeit an.

- Datenbereitstellung mit und ohne Delay

  Hierbei handelt es sich eigentlich nicht um einen Leistungsengpass. Allerdings wurde mit der Leistungsanalyse festgestellt, dass die unterschiedliche Bereitstellung der Flugzeugnachrichten in den beiden Delay-Einstellung im *ATC.Server* zu einem unterschiedlichen Leistungsverhalten führt. So konnte generell eine größere Benutzerlast verarbeitet werden, wenn für diese bei der Datenbereitstellung im *ATC.Server* kein Delay berücksichtigt werden musste. Die Ursachen für dieses unterschiedliche Verhalten konnte ohne Einsatz eines Profiling Tools leider nicht ermittelt. Wahrscheinlich ist aber, dass das unterschiedliche Leistungsverhalten auf die Berechnung der bereitzustellenden Flugzeugnachrichten bei der Berücksichtigung des Delays zurückzuführen ist.

5 Optimierung

Im Rahmen dieser Bachelor-Thesis besteht zusätzlich zur Leistungsanalyse die Aufgabe Optimierungen am Flugdatenserver selber vorzunehmen bzw. Möglichkeiten zur Optimierung aufzuzeigen. Nachfolgend werden einige Möglichkeiten zur Optimierung des Flugdatenservers dargestellt. Aus zeitlichen Gründen wurde im Rahmen dieser Arbeit nur eine dieser Optimierungen umgesetzt.

5.1 Verteilung auf getrennte Systeme

Eine einfache Möglichkeit die Leistungsfähigkeit des Flugdatenservers zu erhöhen besteht darin, die beiden Flugdatenserver-Prozesse auf getrennten Systemen auszuführen. In Abbildung 5.1.1 wird dies exemplarisch dargestellt. Dadurch dass die beiden Prozesse auf verschiedenen Systemen ausgeführt werden, hat jeder Prozess prinzipiell mehr CPU-Leistung zur Verfügung. Geht man davon aus, dass die beiden Systeme über dieselbe Ressource (CPU) verfügen wie der aktuelle Flugdatenserver, kann je nach Konfiguration die doppelte Benutzerlast verarbeitet werden.

Aktuell teilen sich die beiden Flugdatenserver-Prozesse die verfügbare CPU-Leistung auf demselben System. Wie aus der Analyse des Gesamtsystems in Abschnitt 4.4.2 hervorgeht, wird durch eine Aufteilung der Flugdatenserver-Prozesse auf separate Systeme auch die enorme Anzahl an Kontextwechseln, was eine CPU entsprechend belastet, auf die beiden Systeme verteilt werden. Generell kann die gesamte Ressourcenverwendung auf zwei Systeme aufgeteilt werden, was entsprechend mehr Leistung für das Gesamtsystem bedeutet.

Abb. 5.1.1 Ausführung der Flugdatenserver-Prozesse auf getrennten Systemen.

5.2 Load Balancing

Mit *Load Balacing* wird allgemein eine Lastverteilung auf mehrere parallel arbeitende Systeme bezeichnet, um damit eine Überlastung eines Systems zu verhindern.

Auf den Flugdatenserver bezogen besteht die Möglichkeit die Benutzerlast auf mehrere parallel laufende *ATC.Webserver* zu verteilen. Diese wiederum nutzen einen oder ebenfalls mehrere parallel laufende *ATC.Server*.

Für das *Load Balancing* gibt es Software- und Hardwaretechnische Lösungen, auf die im Rahmen dieser Arbeit jedoch nicht eingegangen wird. In der unten stehenden Abbildung 5.2.1 wird das *Load Balancing* exemplarisch als *Load Balancer* dargestellt. Grundsätzlich realisiert der *Load Balancer* die Weiterleitung der eingehenden Client-Anfragen auf die verschiedenen *ATC.Webserver*, abhängig von deren Auslastung. Wie bereits erwähnt, können zudem mehrere *ATC.Server* verwendet werden, die dann von den *ATC.Webservern* genutzt werden. Mit diesem Verfahren könnten potenziell große Mengen an Benutzern verarbeitet werden.

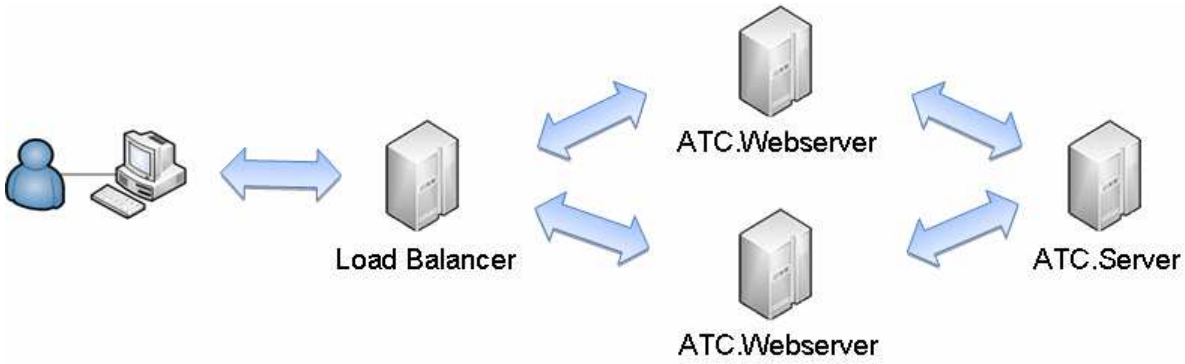


Abb. 5.2.1 Load Balacing zur Lastverteilung des Flugdatenservers.

5.3 Graceful Degredation

Mit *Graceful Degredation* wird ein Mechanismus bezeichnet, der es dem Flugdatenserver selbst ermöglicht im Überlastfall automatisch eine Reaktion auf die Überlastung durchzuführen, um die Überlast zu reduzieren. Eine solche Reaktion könnte im einfachsten Fall bei Überschreiten einer bestimmten Ressourcen-Auslastung das Ablehnen weiterer anfragender Benutzer sein.

Mit den Erkenntnissen aus der Leistungsanalyse, lassen sich aber weitaus intelligentere Mechanismen entwickeln. Wie aus der Analyse hervorging kann, mit einer höheren Update-Rate (> 1000 ms) generell eine größere Benutzerlast verarbeitet werden. Diese Erkenntnis kann für einen *Graceful Degredation* Mechanismus genutzt werden. So könnte man diesen Mechanismus so gestalten, dass im Überlastfall die Update-Rate, mit der die Client-Anfragen erfolgen, schrittweise so lange erhöht wird, bis die Überlast reduziert bzw. aufgehoben ist. Hat sich die Auslastung dann über einen gewissen Zeitraum „erholt“, kann die Update-Rate automatisch wieder gesenkt werden. Als Indikator, um eine Überlast feststellen zu können, können beispielsweise die Leistungsmetriken *CPU-Auslastung* und *ASP.NET Anfragen pro Sekunde* verwendet werden.

5.4 Algorithmische Optimierungen

Anhand der Erkenntnisse aus der Leistungsanalyse gibt es einige Möglichkeiten die Implementierung des Flugdatenservers algorithmisch zu optimieren. Nachfolgend werden einige Möglichkeiten aufgezeigt, an welchen Stellen der Implementierung des Flugdatenservers Optimierungen integriert werden könnten.

- Web-Service JSON-Format

  Wie aus der Leistungsanalyse hervorging, ist das JSON-Format für die Übertragung der Flugzeugnachrichten vom Web-Service zum Web-Client nicht so günstig, aufgrund der im Vergleich zum XML-Format längeren Serialisierungszeiten. Hier macht es durchaus Sinn die Flugzeugnachrichten im XML-Format zu übertragen, um kürzere Serialisierungszeiten und damit Verarbeitungszeiten auf dem Flugdatenserver zu erhalten. Allerdings muss man sich dann im Klaren sein, dass die Menge der zu übertragenden Daten durch das XML-Format entsprechend steigt. Zudem müssen im Web-Client die erhaltenen XML-Daten dann entsprechend in JavaScript-Objekte umgewandelt werden.

- Bereitstellung der Flugzeugnachrichten auf dem *ATC.Server*

  Mit der Leistungsanalyse wurden leichte Unterschiede bei der Datenbereitstellung zwischen den beiden Delay-Einstellungen (mit und ohne) festgestellt. So konnte mit Delay eine weitaus geringere Benutzerlast als ohne Delay verarbeitet werden. Wie schon beschrieben, liegt dies vermutlich an der Berechnung der bereitzustellenden Menge an Flugzeugnachrichten, die unter Verwendung eines Delay erfolgt. Wie in Kapitel 2.2.4

beschrieben, erfolgt hierfür ein Vergleich von Zeitstempeln von einer bestimmten Menge an Flugzeugnachrichten.

Eine Möglichkeit dieses Verfahren zu optimieren, wäre es, die Anzahl der Vergleiche von Zeitstempeln um ein Vielfaches zu reduzieren. Aktuell wird jede eingehende Flugzeugnachricht mit dem aktuellen Zeitstempel versehen, womit dann die Vergleiche zur Ermittlung der Delay-Position erfolgen. Wie in Abschnitt 2.1.3 beschrieben, liefert eine Antenne bzw. das verwendete *ASTERIX CAT 21* Protokoll mehrere Flugzeugnachrichten in einem UDP-Datagramm. Dadurch wäre es möglich, die mit einem UDP-Datagramm erhaltenen Flugzeugnachrichten mit einem Container-Objekt zusammenzufassen, welches dann mit einem einzigen Zeitstempel versehen wird. Die Vergleiche der Zeitstempel erfolgen dann nicht auf Basis jeder Flugzeugnachricht, sondern auf Basis des Container-Objektes. Wenn man davon ausgeht, dass eine *ASTERIX CAT 21* Nachricht zwischen 4 und 6 Flugzeugnachrichten liefert, dann lassen sich die Vergleiche der Zeitstempel dementsprechend um das 4- bzw. 6-fache reduzieren.

- Reduzierung der Flugzeugnachrichten im *ATC.Server*

  Vorab ist zu erwähnen, dass diese algorithmische Optimierung im Rahmen dieser Arbeit realisiert wurde. Wie in Abschnitt 2.2.4 beschrieben, ist im *Privilegierten-Web-Service* eine Methode implementiert, die die Menge an Flugzeugnachrichten, die an einen Web-Client zurückgegeben wird, reduziert. Wie beschrieben, wurde diese Optimierung bereits vor dieser Bachelor-Thesis im *Privilegierten-Web-Service* implementiert. Anhand der Leistungsanalyse in Abschnitt 4.4.2, konnte das Ergebnis dieser Optimierung nachgewiesen werden. So konnte mit der Nutzung des *Privilegierten-Web-Service* weitaus größere Benutzerlasten verarbeitet werden, als über den *Standard-Web-Service*. Dieser Unterschied resultierte aus der unterschiedlichen Menge an Flugzeugnachrichten, die an den Web-Client zurückgegeben werden und der damit verbundenen Problematik der Serialisierung der Flugzeugnachrichten in das JSON-Format (siehe Abschnitt 4.4.2, Leistungsmetrik 3 und 5).

  Damit die Menge der Flugzeugnachrichten nicht nur für den *Privilegierten-Web-Client* sondern auch für den *Standard-Web-Client* reduziert wird, ist es erforderlich, den Aufruf dieser Methode an eine andere Stelle zu platzieren, also nicht innerhalb des *Privilegierten-Web-Service*. Der Methodenaufruf wurde also im *Privilegierten-Web-Service* entfernt und stattdessen im ATC.Server platziert. In den nachfolgenden Listings 5.4.1 und 5.4.2 wird die Optimierung der Implementierung noch einmal mit Pseudocode verdeutlicht. Die

Änderungen wurden jeweils rot markiert bzw. durchgestrichen. Der Pseudocode der ursprünglichen Implementierung kann zudem aus Abschnitt 2.2.4 hinzugezogen werden.

```
ATC.WebServer

public class StdWebService
{
      public List<Airplane> getAirplane(seqNo, selectedAirplane)
      {
            return AtcServer.getAirplane(seqNo, selectedAirplane)
      }
}


public class PrivWebService
{
      public List<Airplane> getAirplane(source, seqNo, selectedAirplane)
      {
            List<Airplane> res = AtcServer.getAirplane(seqNo, source,
selectedAirplane);
            ~~res = reduceAirplanes(res, selctedAirplane)~~
            return res;
      }
```

Listing 5.4.1: Pseudocode für die beiden Web-Services auf dem *ATC.Webserver*.

```
ATC.Server

public class DataSourceMgmt
{
      public List<Airplane> getAirplane(seqNo, selectedAirplane)
      {
            lastPosition = buffer.getLastPosition();

            If seqNo == -1
                  seqNo = getPosition(lastPosition, updateRate + delay);

            delayPosition = getPosition(lastPosition, delay);

            List<Airplane> res = buffer.getAirplane(seqNo, delayPosition);
            return reduceAirplanes(res, selectedAirplane);
      }

      public List<Airplane> getAirplane(seqNo, antenna, selectedAirplane)
      {
            lastPos = buffer.getLastPosition();

            If seqNo == -1
                  seqNo = getPosition(lastPosition, updateRate)

            List<Airplane> res = buffer.getAirplane(seqNo, lastPos,
            antenna);
            return reduceAirplanes(res, selectedAirplane);
      }
}
```

Listing 5.4.2: Pseudocode für die beiden Methoden zur Datenbereitstellung auf dem *ATC.Server*.

6 Zusammenfassung

Im Rahmen dieser Bachelor-Thesis wurde der Flugdatenserver einer intensiven Leistungsanalyse unterzogen. Für diesen Zweck wurde geprüft, welche Möglichkeiten und Techniken vorhanden waren, um eine Leistungsanalyse durchführen zu können. Des Weiteren wurde eine eigene verteilte Anwendung zur Leistungsanalyse entwickelt. Diese verteilte Anwendung realisierte zum einen eine Simulation einer beliebig hohen Benutzerlast sowie die Messung von konfigurierbaren Leistungsmetriken. Mit der Simulation der Benutzerlast konnte eine beliebige Anzahl an Benutzern simuliert werden, die den Flugdatenserver nutzten. Mit Hilfe dieser verteilten Anwendung wurde die Leistungsanalyse des Flugdatenservers durchgeführt.

Mit der Leistungsanalyse selber wurde das Leistungsverhalten des Flugdatenservers mit einer steigenden Benutzerlast ermittelt. Zudem konnten damit die maximalen Benutzerlasten, die unter verschiedenen Konfigurationen verarbeitet werden können, ermittelt werden. Weiterhin wurden einige Leistungsengpässe innerhalb des Flugdatenservers aufgedeckt. Mit den Ergebnissen aus der Leistungsanalyse konnte geklärt werden, inwiefern die Flugdatenserver Web-Anwendung einer großen Menge an Benutzern zur Verfügung gestellt werden kann.

Anschließend wurden auf Basis der Ergebnisse der Leistungsanalyse eine Optimierung umgesetzt und Hinweise auf weitere Möglichkeiten zur Optimierung gegeben, um die Leistung des Flugdatenservers zu erhöhen.

Mit der verteilten Anwendung wurde ein Programm zur Leistungsanalyse entwickelt, dass für weitere Analysen eingesetzt werden kann.

7 Ausblick

Wie einleitend in dieser Bachelor-Thesis beschrieben, soll der Flugdatenserver im Rahmen der Internet-Präsenz einer breiten Masse an Benutzern zur Verfügung gestellt werden. Nach den Ergebnissen aus der Leistungsanalyse ist das allerdings derzeit nur bedingt möglich. Mit der Umsetzung der in Kapitel 5 genannten Optimierungen kann die Leistungsfähigkeit des Flugdatenservers je nach Optimierung aber entsprechend gesteigert werden.

Die Umsetzung der jeweiligen Optimierung ist natürlich davon abhängig, wie viel Aufwand für das Erreichen einer bestimmten Leistungsfähigkeit investiert werden kann. Soll die Flugdatenserver Web-Anwendung nur einer bestimmten Menge an Benutzer, z.B. allen Studenten der Hochschule Offenburg, zur Verfügung gestellt werden, dann sind einfache algorithmische Optimierungen an der Implementierung oder die Aufteilung der Flugdatenserver-Prozesse auf getrennte Systeme ausreichend. Soll die Web-Anwendung aber über das Internet einer wirklich großen Benutzermenge zur Verfügung gestellt werden, ist die Umsetzung eines *Load Balancing* Mechanismus unumgänglich.

Tabellenverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[MEI04] J.D. Meier, Srinath Vasireddy, Ashish Babbar, Alex Mackman, Improving .NET Application Performance and Scalability: Patterns an Practices, Microsoft Corporation, 2004, ISBN 0-7356-1851-8

[GLAV10] Paul Glavich, Chris Farrell, .NET Performance Testing and Optimization - The Complete Guide, redgate books, 2010, ISBN 978-1-906434-40-3

[MEN04] Heinrich Mensen, Moderne Flugsicherung, 3. Auflage, Springer Verlag, 2004, ISBN 3-540-20581-0

[PAH09] Simon Pahl, Bachelor-Thesis WS 2008/2009, Entwicklung eines 3D Darstellungs-systems für Flugsicherungsdaten auf Basis von Google Earth, 2009

[CAT21] EUROCONTROL, EUROCONTROL Standard Document for Surveillance Data Exchange Part 12 : Category 21

[DEV01] Performance Comparison: SOAP vs. JSON (WCF-Implementaiton) [Online], http://developers.de/blogs/damir_dobric/archive/2008/12/27/performance-comparison-soap-vs-json-wcf-implementation.aspx

[WikFlug] Flugsicherung, Wikipedia, [Online], 2011, http://de.wikipedia.org/wiki/Flugsicherung

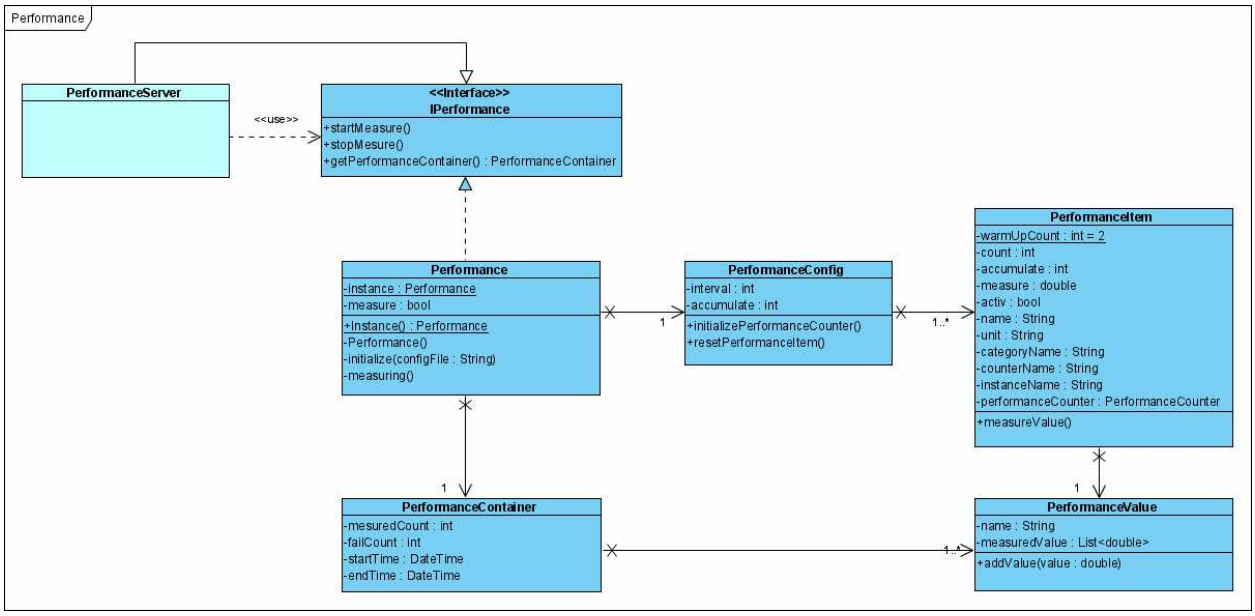
[WikPri] Primärradar, Wikipedia, [Online], 2011, http://de.wikipedia.org/wiki/Prim%C3%A4rradar

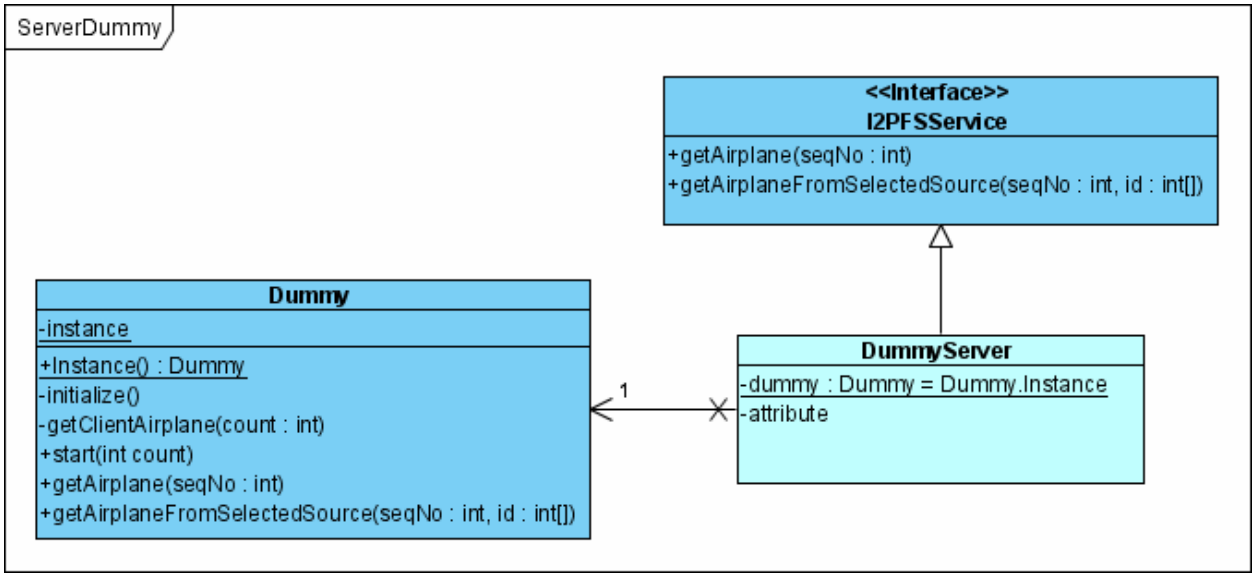
[WikSek] Sekundärradar, Wikipedia, [Online], 2011 http://de.wikipedia.org/wiki/Prim%C3%A4rradar

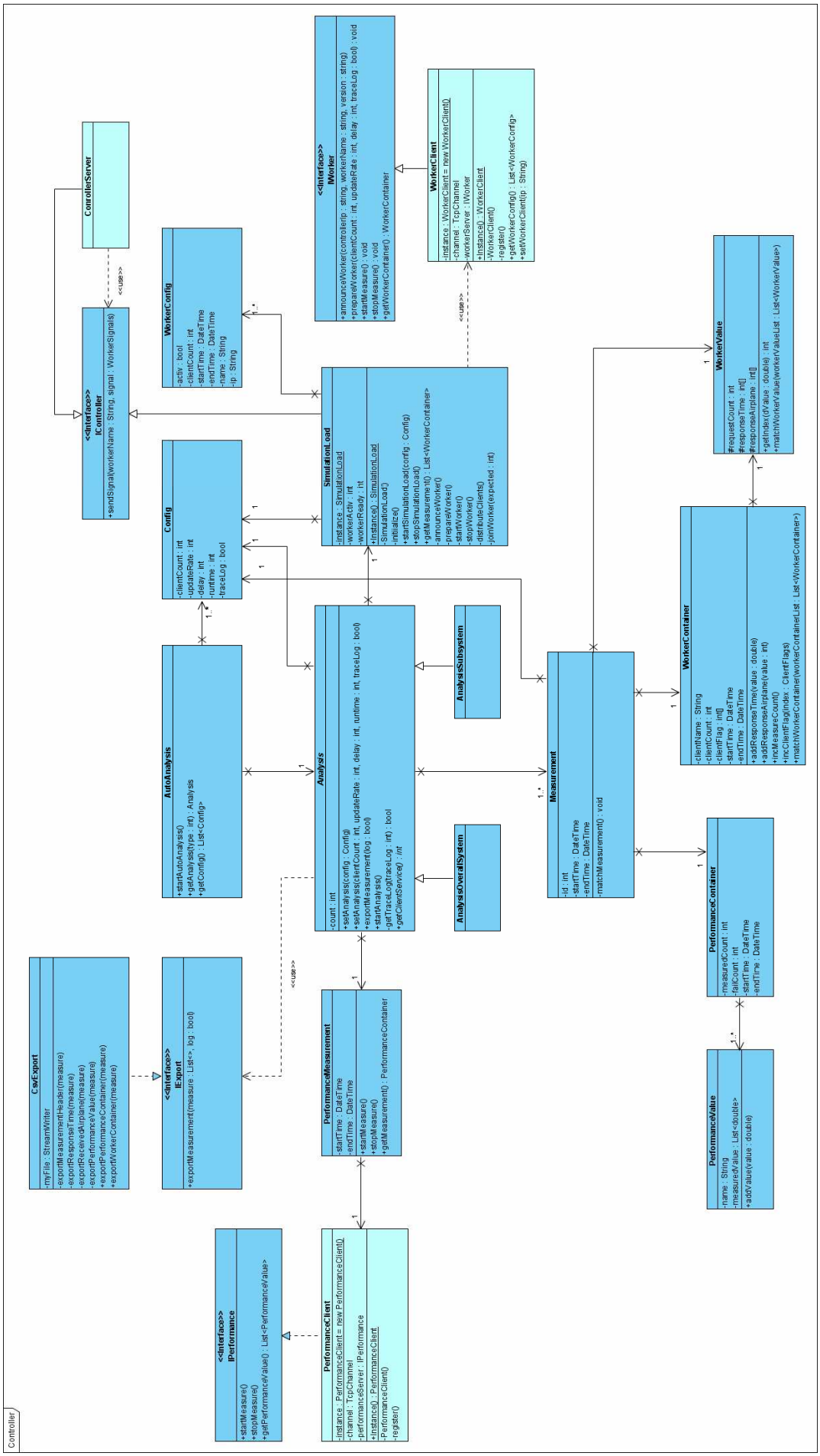
[WikAds] Automatic Dependent Surveillance, , Wikipedia, [Online], 2011 http://de.wikipedia.org/wiki/Automatic_Dependent_Surveillance

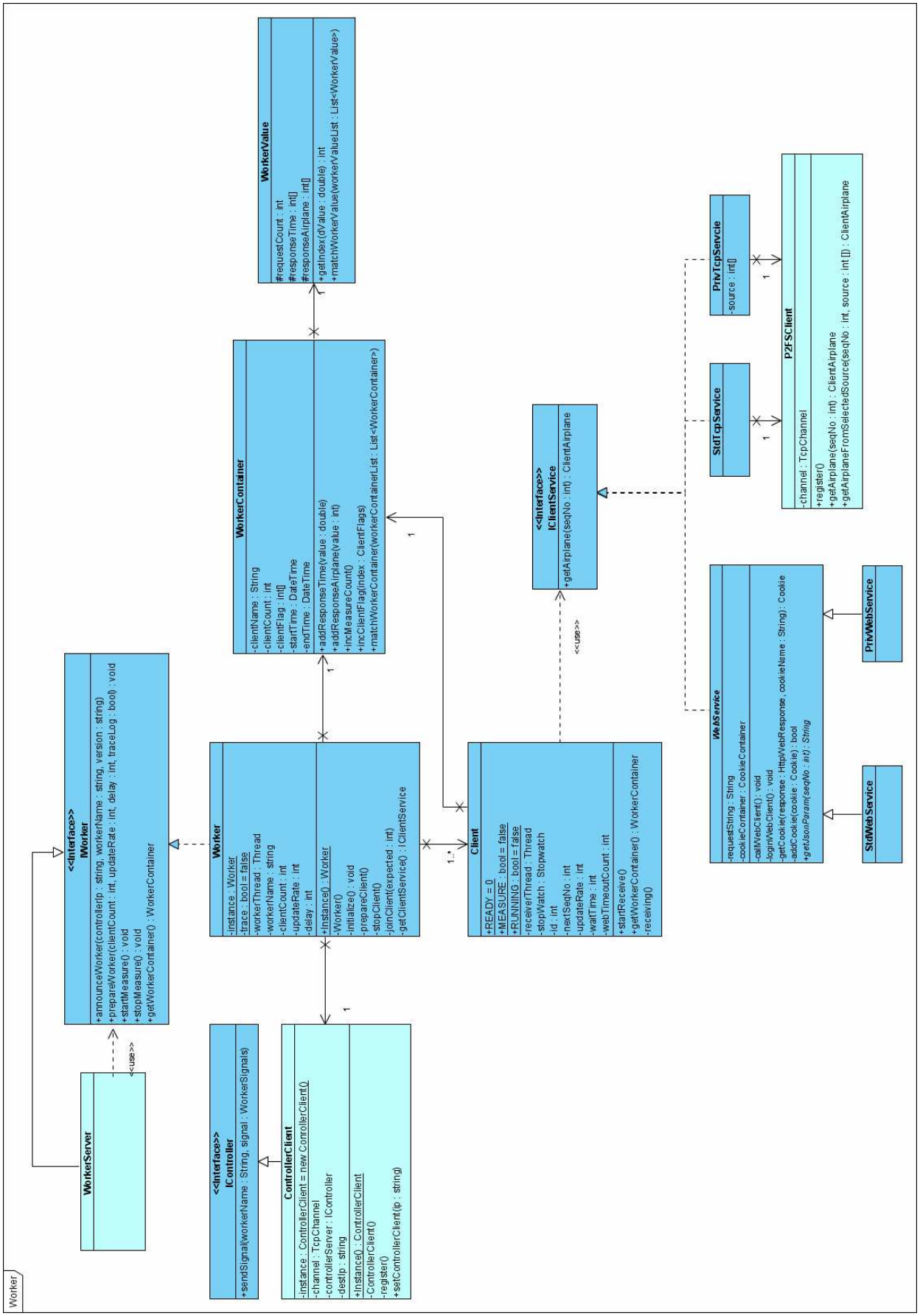
Anhang

Anhang A - UML-Klassendiagramme






Controller
ControllerServer
<<Interface>>
IController
+sendSignal(workerName : String, signal : WorkerSignals)
<<use>>
WorkerConfig
-activ : bool
-clientCount : int
-startTime : DateTime
-endTime : DateTime
-name : String
-ip : String
<<Interface>>
IWorker
+announceWorker(controllerIp : string, workerName : string, version : string)
+prepareWorker(clientCount : int, updateRate : int, delay : int, traceLog : bool) : void
+startMeasure() : void
+stopMeasure() : void
+getWorkerContainer() : WorkerContainer
WorkerClient
-instance : WorkerClient = new WorkerClient()
-channel : TcpChannel
-workerServer : IWorker
+Instance() : WorkerClient
-WorkerClient()
-register()
+getWorkerConfig() : List<WorkerConfig>
+setWorkerClientIp : String)
SimulationLoad
-instance : SimulationLoad
-workersActiv : int
-workersReady : int
+Instance() : SimulationLoad
-SimulationLoad()
-initialize()
+startSimulationLoad(config : Config)
+stopSimulationLoad()
+getMeasurement() : List<WorkerContainer>
-announceWorker()
-prepareWorker()
-startWorker()
-stopWorker()
-distributeClients()
-joinWorker(expected : int)
Config
-clientCount : int
-updateRate : int
-delay : int
-runtime : int
-traceLog : bool
WorkerValue
#requestCount : int
#responseTime : int[]
#responsesAirplane : int[]
+getIndex(dValue : double) : int
+matchWorkerValue(workerValueList : List<WorkerValue>)
AutoAnalysis
+startAutoAnalysis()
+getAnalysis(type : int) : Analysis
+getConfig() : List<Config>
Analysis
-count : int
+setAnalysis(config : Config)
+setAnalysis(clientCount : int, updateRate : int, delay : int, runtime : int, traceLog : bool)
+exportMeasurement(log : bool)
+startAnalysis()
-getTraceLog(traceLog : int) : bool
+getClientService() : int
AnalysisSubsystem
AnalysisOverallSystem
Measurement
-id : int
-startTime : DateTime
-endTime : DateTime
-matchMeasurement() : void
WorkerContainer
-clientName : String
-clientCount : int
-clientFlag : int[]
-startTime : DateTime
-endTime : DateTime
+addResponseTime(value : double)
+addResponseAirplane(value : int)
+incMeasureCount()
+incClientFlag(index : ClientFlags)
+matchWorkerContainer(workerContainerList : List<WorkerContainer>)
PerformanceContainer
-measuredCount : int
-failCount : int
-startTime : DateTime
-endTime : DateTime
PerformanceValue
-name : String
-measuredValue : List<double>
+addValue(value : double)
CsvExport
-myFile : StreamWriter
-exportMeasurementHeade(measure)
-exportResponseTime(measure)
-exportReceivedAirplane(measure)
-exportPerformanceValue(measure)
+exportPerformanceContainer(measure)
+exportWorkerContainer(measure)
<<Interface>>
IExport
+exportMeasurement(measure : List<>, log : bool)
PerformanceMeasurement
-startTime : DateTime
-endTime : DateTime
+startMeasure()
+stopMeasure()
+getMeasurement() : PerformanceContainer
<<Interface>>
IPerformance
+startMeasure()
+stopMeasure()
+getPerformanceValue() : List<PerformanceValue>
PerformanceClient
-instance : PerformanceClient = new PerformanceClient()
-channel : TcpChannel
-performanceServer : IPerformance
+Instance() : PerformanceClient
-PerformanceClient()
-register()


Worker
WorkerServer
<<use>>
<<Interface>>
IWorker
+announceWorker(controllerIp : string, workerName : string, version : string)
+prepareWorker(clientCount : int, updateRate : int, delay : int, traceLog : bool) : void
+startMeasure() : void
+stopMeasure() : void
+getWorkerContainer() : WorkerContainer
<<Interface>>
IController
+sendSignal(workerName : String, signal : WorkerSignals)
ControllerClient
-instance : ControllerClient = new ControllerClient()
-channel : TcpChannel
-controllerServer : IController
-destIp : string
+Instance() : ControllerClient
-ControllerClient()
-register()
+setControllerClient(ip : string)
Worker
-instance : Worker
-trace : bool = false
-workerThread : Thread
-workerName : string
-clientCount : int
-updateRate : int
-delay : int
+Instance() : Worker
-Worker()
-initialize() : void
-prepareClient()
-stopClient()
-joinClient(expected : int)
-getClientService() : IClientService
WorkerContainer
-clientName : String
-clientCount : int
-clientFlag : int[]
-startTime : DateTime
-endTime : DateTime
+addResponseTime(value : double)
+addResponseAirplane(value : int)
+incMeasureCount()
+incClientFlag(index : ClientFlags)
+matchWorkerContainer(workerContainerList : List<WorkerContainer>)
WorkerValue
#requestCount : int
#responseTime : int[]
#responseAirplane : int[]
+getIndex(dValue : double) : int
+matchWorkerValue(workerValueList : List<WorkerValue>)
Client
+READY = 0
+MEASURE : bool = false
+RUNNING : bool = false
-receiverThread : Thread
-stopWatch : Stopwatch
-id : int
-nextSeqNo : int
-updateRate : int
-waitTime : int
-webTimeoutCount : int
+startReceive()
+getWorkerContainer() : WorkerContainer
-receiving()
<<Interface>>
IClientService
+getAirplane(seqNo : int) : ClientAirplane
<<use>>
WebService
-requestString : String
-cookieContainer : CookieContainer
-callWebClient() : void
-loginWebClient() : void
-getCookie(response : HttpWebResponse, cookieName : String) : Cookie
-addCookie(cookie : Cookie) : bool
+getJsonParam(seqNo : int) : String
PrivWebService
StdWebService
PrivTcpService
-source : int[]
StdTcpService
P2FSClient
-channel : TcpChannel
+register()
+getAirplane(seqNo : int) : ClientAirplane
+getAirplaneFromSelectedSource(seqNo : int, source : int[]) : ClientAirplane
1
1..*

Anhang B – Lastenheft

Lastenheft - Leistungsanalyse und Optimierung eines Flugdatenservers

Autor des Dokuments	Francois Hilberer	Erstellt am	25.03.2011
Dateiname	Lastenheft.doc		
Seitenanzahl	6	© 2011	

Inhaltsverzeichnis

1 Ausgangssituation und Zielbestimmung

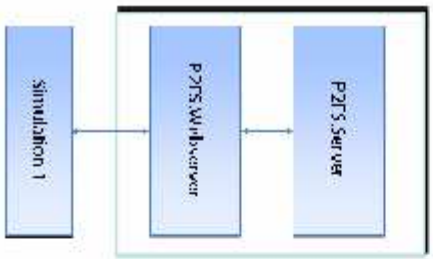
Für den in Projekt 2 entwickelten Flugdatenserver soll eine Leistungsanalyse vorgenommen werden um mögliche Bottlenecks zu identifizieren und zu beheben.

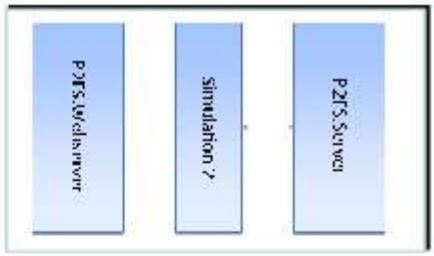
Der Flugdatenserver besteht aus zwei Komponenten, dem P2FS.Server und dem P2FS.Webserver. Der P2FS.Server realisiert unter anderem den Datenempfang von den Antennen und die Weitergabe der Daten an die Web-Anwendung bzw. Webserver. Der Webserver stellt für die Clients eine Web-Anwendung zur Verfügung.

Die Leistungsanalyse wird mit Hilfe einer Simulation der Zugriffe auf den Flugdatenserver durchgeführt. Die Simulation kann wahlweise das Gesamtsystem also P2FS.Server u. P2FS.Webserver (Simulation 1) oder den P2FS.Server als einzelne Komponente testen (Simulation 2). Weiterhin erfolgt unter Verwendung von Simulation 1 ein Hüllentest, der beispielsweise nur Dummy Daten weiterreicht. Geplant ist auch der Einsatz eines Profiling-Tools das unter anderem die Laufzeiten der einzelnen Methoden ermittelt.

Mit der Ausführung der oben beschriebenen Simulationen bzw. Tools können Engpässe ermittelt und die verantwortlichen Komponenten bzw. Module identifiziert werden. Durch Ermittlung und Auswertung system- u. domänenspezifischer Eigenschaften wie CPU-, Speicher-, Netzwerk Interface- Auslastung und Response Antwortzeiten des Servers ist eine exaktere Analyse möglich.

Ziel ist es die Web-Anwendung des Flugdatenservers einer breiten Masse zur Verfügung zu stellen, ohne dass dabei Performance Engpässe auftreten.





2 Anforderungen Leistungsanalyse

2.1 Musskriterien

/Req 2.1-10/ Für eine Simulation kann die Anzahl der simulierten Clients konfiguriert werden.

/Req 2.1-20/ Für eine Simulation kann die Updaterate konfiguriert werden. Die Updaterate ist für alle simulierten Clients gleich.

/Req 2.1-30/ Für eine Simulation kann eine Delay-Einstellung (mit oder ohne Delay) konfiguriert werden.

/Req 2.1-40/ Die simulierten Clients werden mit einem 200 msec (ggfs. + Zufallswert) Intervall bzw. Zeitdifferenz zu unterschiedlichen Zeitpunkten gestartet.

/Req 2.1-50/ Eine Simulation kann manuell gestartet bzw. gestoppt werden.

/Req 2.1-60/ Die simulierten Clients müssen sich aus Sicht des Servers in allen für die Leistungsmessung relevanten Aspekten wie normale Clients verhalten.

/Req 2.1-70/ Während der Durchführung einer Simulation werden die aktuellen Flugdatenserver Systemeigenschaften CPU- und Speicher Auslastung prozessbezogen (P2FS.Server u. P2FS.Webserver) ermittelt. Die Intervall-Länge bzw. Granularität der Messwertaufnahme ist im Bereich von 25 - 5000 msec konfigurierbar.

/Req 2.1-80/ Während einer Simulation werden die Bytes/sec innerhalb eines repräsentativen Intervalls ermittelt. Auf dieser Basis sind Rückschlüsse auf die Netzwerk Auslastung möglich.

/Req 2.1-90/ Während einer Simulation werden die Antwortzeiten des Servers (Response) auf die simulierten Client Anfragen (Request) ermittelt und kategorisiert. Die Kategorisierung erfolgt in Form von „1-200 msec -- 100 Responses", „201-500msec -- 40 Responses" usw.

/Req 2.1-100/ Nach Ende einer Simulation können die ermittelten system- u. domänenspezifischen Eigenschaften (CPU-, Speicher- u. Netzwerk Auslastung, Antwortzeiten des Servers) als csv Format exportiert werden.

/Req 2.1-110/ Die Simulation kann wahlweise das Gesamtsystem (P2FS.Server u. P2FS.Webserver) oder den P2FS.Server als einzelne Komponente testen.

/Req 2.1-120/ Die Simulationsumgebung darf nicht zum Bottleneck werden.

2.2 Kannkriterien

/Req 2.2-10/ Abhängig von /Req 2.1-120/ kann die Simulation mittels einem verteilten Algorithmus auf mehreren Rechnern ausgeführt werden, wobei die Auswertung auf einem Initiator Rechner stattfindet.

3 Strategie Leistungsanalyse

/Req 3.10/ Ausführung der Simulationen (Simulation1, Simulation2, Hüllentest) zur Generierung von Messreihen auf Basis unterschiedlicher Konfigurationen

/Req 3.20/ Ausführung der Simulationen im Netzwerk ggf. auch lokal (ohne Netzwerk und Firewall)

/Req 3.30/ Einsatz eines Profiling-Tool

4 Mögliche Optimierungen

Im Voraus sind bereits folgende Optimierungen angedacht, die jedoch von dem Ergebnis der Leistungsanalyse abhängig sind.

/Req 4.10/ P2FS.Server und P2FS.Webserver auf getrennte Server bzw. Hardware

/Req 4.20/ Algorithmische Optimierungen (z.B. DSM Speichermanagement)

/Req 4.30/ „load balancing" (Verteilung Anfragelast auf mehrere Server)

/Req 4.40/ „graceful degradation" (automatische Reduktion der Update-Rate im Überlastfall)

5 Entwicklungsumgebung

/Req 5.10/ Visual Studio 2008 SP1

/Req 5.20/ .NET Framework 3.5